# Improving the Compiler/Software DSM Interface: Preliminary Results

Pete Keleher   Chau-Wen Tseng
*keleher@cs.umd.edu*   *tseng@cs.umd.edu*
Dept. of Computer Science
University of Maryland
College Park, MD 20742

## Abstract

Current parallelizing compilers for message-passing machines only support a limited class of data-parallel applications. One method for eliminating this restriction is to combine powerful shared-memory parallelizing compilers with software distributed-shared-memory (DSM) systems. Preliminary results show simply combining the parallelizer and software DSM yields very poor performance. The compiler/software DSM interface can be improved based on relatively little compiler input by: 1) combining synchronization and parallelism information communication on parallel task invocation, 2) employing customized routines for evaluating reduction operations, and 3) selecting a hybrid update protocol to presend data by flushing updates at barriers. These optimizations yield decent speedups for program kernels, but are not sufficient for entire programs. Based on our experimental results, we point out areas where additional compiler analysis and software DSM improvements are necessary to achieve good performance.

## 1 Introduction

Though microprocessor speeds are continuing to increase, most observers agree that parallel computing represents the only plausible way to significantly increase the computational power available. Parallel machines range from multiprocessor workstations (e.g., SGI PowerChallenge, DEC Sable) to scalable message-passing distributed-memory systems (e.g., IBM SP-2, Intel Paragon). Despite their promise, however, parallel computers are not likely to be widely successful until they are easy to use. A key problem in parallel computing is to provide a portable and easy method for programming multiprocessors, particularly large message-passing machines. There have been many approaches for overcoming this important obstacle, but they have limitations in either usability or performance.

One approach is to write explicitly parallel programs, using parallel dialects of Fortran or C which provide DOALL and PARALLEL DO annotations that users can easily add to indicate parallelism. However, performance can be poor unless users extensively rewrite programs to avoid problems such as poor spatial locality and false sharing [32]. For distributed-memory machines, users can write message-passing programs that use a standard message-passing library such as PVM, P4, PARMACS, or MPI. Users can achieve high performance because they have total control over interprocessor communication and data layout. Message-passing programs, however, must deal with separate address spaces, index translation, and explicit interprocessor communication. Writing efficient parallel programs thus require too much effort for most scientists and engineers.

Another solution is to use data-parallel languages such as High Performance Fortran (HPF) [22]. HPF is an enhanced Fortran 90 extended with annotations that specify how data should be partitioned across processors. Compilers have been developed (e.g., Fortran D [19], Paradigm [33]) that can translate HPF programs into message-passing programs for distributed-memory machines. Extensive compiler and run-time support (e.g., Chaos [10]) have also been developed to handle programs with complicated reference patterns, such as those found in adaptive sparse applications.

Though HPF is a good solution for data-parallel applications, there are still a number of disadvantages to using HPF. First, users are forced to rewrite their application using the data-parallel constructs found in Fortran 90. Though the process is not as difficult as writing message-passing code, it may still be laborious for large legacy codes. Some compilers for distributed-memory machines can avoid this problem by automatically detecting data-parallelism in sequential programs (e.g., Fortran D [19]). More problematic is the fact that some applications may contain irregular data access patterns or parallelism not expressible in the data-parallel constructs found in HPF, such as trees and linked lists found in C. Current data-parallel languages and compilers for distributed-memory machines are thus limited in their applicability, since precise information is needed.

### 1.1 Shared-memory Compilers and Software DSMs

Instead of relying on explicitly parallel programs (high effort) or data-parallel compilers (limited applicability), we suggest another approach for programming message-passing machines based on combining shared-memory compilers and software DSMs. Evidence indicates parallelizing compilers for shared-memory machines are beginning to mature. Several research prototypes have been developed with powerful symbolic and interprocedural analyses that can automatically exploit parallelism in many numeric programs (e.g., SUIF [16, 36], Polaris [6]). These compilers generate shared-memory programs with parallel constructs such as DOALL loops and REDUCTION routines.

To exploit shared-memory compilers for message-passing machines, we rely on software distributed-shared-memory (DSM) systems (e.g., Ivy [25], Treadmarks [11]) which support a shared address space using operating systems support. The latest generation of software DSMs (e.g, Munin [4], Blizzard/Tempest [13], CVM [20]) also support multiple coherence protocols and explicit messages on top of existing message-passing machines and networks of workstations.

By combining shared-memory compilers and software DSMs, we create a programming environment that is easy to use, since scientists are no longer required to write their entire programs in data-parallel languages such as HPF. Instead, they can write mostly sequential programs, rewriting a few computation-intensive procedures and adding parallelism directives where necessary. These compilers also have the advantage that they produce programs that can run on the large-scale parallel machines as well as the low-end, but more pervasive workstations. This portability is important for scientists and engineers who want to develop applications that run well on their multiprocessor workstations, but who desire the ability to scale their applications up for larger parallel machines as needed. The combination of ease of use and scalability of software is a key appeal of shared-memory compilers.

A recent development that improves the desirability of compiling for software DSMs is the development of Flexible-Shared-Memory (FSM) machines (e.g, Alewife [1], Flash [18, 23], Typhoon [30, 31]). These architectures maintain a coherent shared address space on top of physically distributed memories, just like traditional shared-memory machines. In addition, FSM machines also support extensible memory coherence protocols and explicit messages where needed to achieve better performance. Because modern software DSMs also support the same features (in software), experience compiling for software DSMs may prove valuable in developing and evaluating new coherence protocols that may be used on new Flexible-Shared-Memory machines. Software DSMs thus can serve as testbeds for future FSM machines.

### 1.2 Contributions

Shared-memory parallelizing compilers are easy to use, flexible, and can accept a wide range of applications. The important question is whether shared-memory compilers targeting software DSMs can approach the performance of current message-passing compilers or explicitly-parallel message-passing programs on distributed-memory machines. This paper provides some preliminary results that attempt to answer this question. We make the following original contributions:

- Demonstrate in a working prototype how the programming model of shared-memory compilers can be combined with the memory system of software DSMs
- Point out the problems with a simple approach to combining shared-memory compilers and software DSMs
- Describe three enhancements to the compiler/software DSM interface: 1) improving parallel task invocation, 2) customized reduction support, and 3) compiler-directed hybrid update protocols
- Experimentally evaluate the performance impact of our enhancements
- Suggest a number of additional improvements based on compiler analysis and software DSM customization.

We begin by considering the parallelization and run-time model of the compiler, the coherence and communication model of the software DSM, and their interactions. We describe three techniques for improving the compiler/software DSM interface. We present our prototype system and some preliminary results, then suggest additional improvements. Following a discussion of related work, we conclude.

# 2 Background

## 2.1 Shared-Memory Compiler Model

The goal of parallelizing compilers is to identify parallel loops or tasks in sequential programs, using data-flow and data dependence analysis combined with program transformations. Computations that occurs frequently in numerical programs are *reductions*, commutative operations (e.g., sum, max) that can be reordered to enable parallelism. Once a parallel portion of the program is identified, it is typically made into the body of a procedure which can be invoked by all the processors in parallel.

Shared-memory parallelizing compilers typically employ a *fork-join* programming model, where a single master thread executes the sequential portions of the program, assigning (forking) computation to additional worker threads when a parallel loop or task is encountered. After completing its portion of the parallel loop, the master waits for all workers to complete (join) before continuing execution. During the parallel computation, the master thread participates by performing a share of the computation just like a worker. After each parallel computation worker threads spin or go to sleep, waiting for additional work from the master thread.

The fork-join model is flexible and can easily handle sequential portions of the computation; however, it imposes two synchronization events per parallel loop. First, a *broadcast barrier* is inserted before the loop body to wake up available worker threads and provide workers with the address of the computation to be performed and parameters if needed. A *barrier* is then inserted after the loop body to ensure all worker threads have completed before the master can continue. Between the broadcast and the barrier threads execute computation in parallel.

Shared-memory parallelizing compilers usually rely on a small run-time system to manage parallelism operations. Typical functions supported in the run-time system include routines for: 1) thread creation at the beginning of the program, 2) assigning parallel computation to workers, 3) performing barrier and lock operations, 4) accumulating the results of global reductions. The run-time system may also support a variety of scheduling policies (e.g., block, round-robin, dynamic) for scheduling iterations of parallel loops to processors.

### 2.2 Software DSM

Software distributed-shared-memory (DSM) systems provide a shared address space on top of physically distributed memory using software support [29]. Efficient implementations have been developed that run on commonly available Unix systems making them widely portable, even to standard UNIX workstations connected via ethernet or ATM networks. Software DSMs rely on (user-level) memory management techniques provided by the operating system to detect accesses and updates to shared data at the granularity of pages. The software DSM system then applys a memory coherence protocol to provide the illusion of shared memory. Simply imitating the coherence protocol used by hardware shared-memory multiprocessors is inefficient due to the high communication overhead and large page-sized coherence units. Techniques developed to improve performance of software DSMs are lazy release consistency and multiple-writer protocols.

**Release consistency.** In the conventional *sequentially consistent* (SC) memory [24] model implemented by most snoopy-cache, bus-based multiprocessors, modifications to shared memory must become visible to other processors immediately [24]. This model is inefficient because it implies communication on each write to a shared data item for which other cached copies exist. In comparison, a *release consistency* (RC) [14] memory consistency model permits a processor to delay making its changes to shared data visible to other processors until special *acquire* or *release* synchronization accesses occur. The propagation of the modifications can thus be postponed until the next synchronization operation takes effect. Programs produce the same results for the two memory models provided that (i) all synchronization operations use system-supplied primitives, and (ii) there is a release-acquire pair between conflicting ordinary accesses to the same memory location on different processors [14]. In practice, most shared-memory programs require little or no modifications to meet these requirements.

**Lazy release consistency.** In *lazy release consistency* (LRC) [21], the propagation of modifications is postponed *until the time of the acquire*. At this time, the acquiring processor determines which modifications it needs to see according to the definition of release consistency. To do so, the execution of each process is divided into *intervals*, each denoted by an *interval index*. Every time a process executes a release or an acquire, a new interval begins and the interval index is incremented. Intervals of different processes are partially ordered by assigning a *vector timestamp* to intervals for each processor. With lazy release consistency, at an acquire, processor $p$ sends its current vector timestamp to the previous releaser, $q$. Processor $q$ then piggybacks on the release-acquire message to $p$ *write notices* for all intervals named in $q$'s current vector timestamp but not in the vector timestamp it received from $p$. Experiments show alternative implementations of release consistency generally cause more communication than lazy release consistency [11].

**Invalid, update, and hybrid protocols.** Write notices indicate that a page has been modified in a particular interval, but do *not* contain the actual modifications. The timing of the actual data movement depends on whether an invalidate, an update, or a hybrid protocol is used [11]. Most DSM systems use an invalidate protocol: the arrival of a write notice for a page causes the processor to invalidate its copy of that page. A subsequent access to that page causes an access miss, at which time the modifications are propagated to the local copy. To keep track of pages, the software DSM keeps a *copyset* for each page listing the processors that have a copy of that page. This set is used to decide which processors need to be informed when the page is modified.

In an *update* coherence protocol, the processor sends a new copy of the data to all the processors in the copyset. This approach is preferable if the processors in the copyset all use the page before new writes, since it eliminates misses. The invalidate protocol is preferable if the processors in the copyset do not use the page, since it avoids unnecessary communication. Experiments show that update protocols generally cause too much wasted communication, since the copyset of a page gradually accumulates processors which no longer need that page. This effect can be countered using a variant of the update protocol called the *hybrid* protocol, which only sends updates for some pages, allowing other pages to be invalidated [11].

**Multiple-writer protocols.** *False sharing* occurs when two or more processors access different variables within a page, with at least one of the accesses being a write. False sharing is problematic for software DSMs because of the large page-size coherence units. *Multiple-writer* coherence protocols [9] avoid false sharing by allowing two or more processors to simultaneously modify their local copy of a shared page. Their modifications are merged at the next synchronization operation. In order to capture the modifications to a shared page, it is initially write-protected. At the first write, a protection violation occurs. The DSM software makes a copy of the page (a *twin*), and removes the write protection so that further writes to the page can occur without any DSM intervention. The twin and the current copy can later be compared to create a *diff*, a runlength encoded record of the modifications to the page. If a new copy of the page is later requested, the processor can send a diff for the page and allow diffs from the multiple writers to be merged by the receiving processor.

Lazy release consistency allows diff creation to be postponed until the modifications are requested, decreasing in the number of diffs created and improving performance. However, *garbage collection* is necessary to reclaim the space used by write notice records, interval records, and diffs. During garbage collection, each processor validates its copy of every page that it has modified. All other pages, all interval records, all write notice records and all diffs are discarded. In addition, each processor updates the copyset for every page.

**Access misses.** To implement consistency, software DSMs usually use the UNIX `mprotect` system call to control access to shared pages. Any attempt to perform a restricted access on a shared page generates a `SIGSEGV` signal. The `SIGSEGV` signal handler examines local information determine the page's state. If the local copy is read-only, the handler allocates a page from the pool of free pages and performs a `bcopy` to create a *twin*. Finally, the handler upgrades the access rights to the original page and returns. If the local page is invalid, the handler requests a copy from a member of the page's approximate copyset. If write notices are present for the page, the faulting processor obtains the list of missing diffs maintained by the system and sends out requests in parallel to all the processors that may have modified the page. When all necessary diffs have been received, they are applied to the page in increasing timestamp order.

# 3 Compiler/Software DSM Interface

From our description of shared-memory compilers and software DSM systems, it should be clear that the two complement each other well. We illustrate the issues involved in the compiler/software DSM interface by listing the steps needed to retarget the Stanford SUIF parallelizing compiler [36] to the CVM software DSM system [20]. We then point out areas where the interface may be improved with relatively little additional compiler analysis.

## 3.1 Simple Interface

A simple way to interface the SUIF compiler and CVM is to port the SUIF run-time system by using routines from CVM for thread startup, locks, and barriers. Some miscellaneous functions for determining logical processor IDs, the total number of threads, and global time also need to be replaced with CVM routines. Since all remaining operations are based on shared memory accesses, they are naturally supported by CVM with no additional effort.

One difficulty that arises is that the SUIF shared-memory compiler assumes a light-weight thread model, where the entire address space is shared by default. Even data allocated on a stack must be made potentially sharable if its address is passed to other processors. In comparison, the CVM software DSM expects a heavy-weight thread model where only memory specifically allocated or marked as shared can be shared between processors. The software DSM cannot simply mark all data in the program as shared, since some data must remain private to each processor.

**Compiler transformations.** To solve this mismatch, the SUIF compiler performs two transformations. First, it promotes all local variables that may be visible to other processors into the global scope. This process includes stack variables whose addresses are passed as parameters to other functions, since they may be used in a parallel region. Actual parameters to functions do cannot and do not need to be promoted to the global scope, since they represent variables declared elsewhere.

The second compiler transformation is to take all global shared variables and make them contiguous in memory by converting them into fields of `suifmem`, a large structured variable (or common block in Fortran). This transformation results in a single global variable containing all of the statically allocated shared memory in the program, as shown in the example below. `suifmem` is padded at both ends to ensure shared and private variables are on separate pages. Space is reserved in `suifmem` for shared data needed by the compiler run-time system, and the compiler generates a variable containing the size of the global variable. During program startup, the run-time system passes the address and size of `suifmem` to the software DSM system, which then marks that region of memory as shared.

```
int A[100];
foo() {
   int B[100];
   forall (i)
      A[i] = B[i];
}
```

```
struct _globmem {
   ...
   int A[100];
   int B[100];
   ...
} _suifmem;

foo() {
   forall (i)
      suifmem.A[i] = suifmem.B[i];
}
```

Note that statically allocated shared memory is important. Because `suifmem` is a statically allocated variable, its address is determined at link time; references to shared data thus take place directly with a compile-time offset. The impact on performance should thus be minor. If shared memory has to be dynamically allocated (e.g., with *shmem_alloc*()), then all accesses to shared memory occur indirectly through a pointer, potentially requiring two memory accesses per shared reference. Multiple source files can be allowed if interprocedural compilation is supported, but in general separate compilation is not possible unless multiple global nonoverlapping global variables are allowed.

### 3.2 Optimizations

The simple interface presented for SUIF and CVM produces a working system, but contains many inefficiencies, some of which may be eliminated with minimal compiler analysis. One of the properties of software DSMs that can lead to poor performance is the use of an *invalidation* protocol for maintaining coherence. Invalidation protocols are preferred because they reduce excessive communication. However, they are inefficient for producer-consumer communication patterns, particularly if there are multiple consumers.

To see why this problem exists, consider what happens when processor $p$ produces data $X$ consumed by processor $q$. By defining $X$, $p$ invalidates the copy of $X$ held by $q$. Using release consistency, the invalidation message is piggybacked on the barrier synchronization message, so there is little overhead for the invalidation. However, when $q$ attempts to consume $X$, it has to take a page fault and wait for the fault handler to initiate a round-trip communication to $p$ to fetch the page containing $X$. If multiple processors need to consume $X$, the producer $p$ is deluged with with a number of requests, adding a serial bottleneck. For certain interconnection networks there may even be contention, reducing performance further.

### 3.2.1 Parallelism Startup

To eliminate these effects, we considered places where producer-consumer relationships occur in compiler-parallelized programs. We consider three opportunities for customizing the software DSM to improve performance. The first is in the parallelism startup code, the portion of the compiler run-time system responsible for awakening worker threads and assigning them work. This operation is a prime example of a producer-consumer relationship, since the master thread produces data (the location of parallel computation to be performed and parameters for the computation) which is consumed by multiple worker threads.

To improve performance for parallelism startup, we enhanced the software DSM to automatically piggyback certain marked locations along with barrier messages. Since the master processor also owns the broadcast barrier preceding each parallel loop, it can combine the broadcast message to the workers acknowledging barrier completion with the information needed for parallelism startup. All that is required is to insert code in the compiler run-time system to mark the section of the global `suifmem` variable reserved for the compiler run-time system. Those variables are then automatically updated with new values with the synchronization messages for the barrier.

### 3.2.2 Customized Reductions

Another opportunity for improving the compiler/software DSM interface is in customized support for reductions. Recall that reductions are commutative actions (e.g., sum, max) identified by the compiler that can be performed on local data and then accumulated into global locations using routines from the compiler run-time library. In the simple system these accumulations are performed as operations on shared memory locations, with lock variables used to guarantee mutual exclusion. In addition to the usual inefficiencies with produce-consumer communication under an invalidation protocol, the need for mutual exclusion in reductions impose a serial bottleneck as well as synchronization traffic for lock acquires and releases.

Fortunately, customized support for reductions can be easily added to a software DSM. The compiler has already identified the operation as a reduction to the run-time system, and the software DSM can take advantage of this information by eliminating lock operations, instead combining the results directly based on each processor's contribution to the accumulated result. The process is simplified because the current SUIF compiler only performs reductions at the end of a parallel region.

CVM's reduction support is implemented by copying the reduction operator and local reduction data into a reduction record. All reduction records are then piggybacked (appended) to the next barrier arrival message to the master thread indicating the worker has completed its portion of the computation. The master thread then performs all the reductions from the last barrier interval, updating the value of the global shared data. The advantage of centralizing the reduction process at the master thread is two-fold. First, synchronization to ensure mutual exclusion is eliminated because the master performs all reductions. Second, since reductions are performed on shared memory, the page containing the reduction data must be valid locally, and a diff describing the reduction is created later. Centralizing the process at the barrier master therefore saves on diff creations, remote misses, and total messages.

### 3.2.3 Hybrid protocol

Finally, we consider the application data communicated between threads during parallel program execution. Good parallelizing compilers such as SUIF typically choose computation partition and loop scheduling policies that promote co-location of data and computation. In loop-intensive numeric codes, the assignment of computation to threads is thus usually fairly stable, yielding consistent sharing patterns for many iterations. By relying on a consistent computation partition, we may be able to obtain a good estimate of communication without doing compile-time analysis by using the copyset information collected by the underlying software DSM system.

Recall that the software DSM keeps track of processors owning a copy of a page in the copyset for that page. This information can be used to improve performance by selectively employing a hybrid invalidate/update coherence protocol. Coherence for pages which are consistently communicated between the same set of processors can be updated rather than invalidated after writes, eliminating access misses. Coherence for the remaining pages is maintained using an invalidate protocol to avoid excessive communication. On the first iteration of the time step loop, the copysets of each page are empty and access misses occur. By the second iteration, however, copyset information indicates the processors that need each page, accurately reflecting stable sharing patterns. Access misses can be then be eliminated by updating processors on the copyset for each page, sending the data before it is accessed.

To evaluate the effectiveness of using a hybrid coherence protocol, we modified the compiler to automatically insert calls to DSM routines that mark pages to be *flushed* at barriers. For a given page, locally modifications are flushed to all other processors in the page's local copyset at each barrier. A processor $p$ is inserted into processor $q$'s copyset for a page if $p$ requests a diff for the page, or if $q$ sees a write notice for the page that was created by $q$.

As previously discussed, barrier flushes of updates (essentially a restricted update model) have both advantages and disadvantages. On the plus side, flushes ideally move data before it is needed, allowing computation and communication to be wholly overlapped. The results can be fewer page invalidations page faults. A second advantage is that lost flush messages do not affect correctness, only performance. Flush messages do not have to reliable, and therefore do not need to be acknowledged. A "flush" therefore consists of only a single message, whereas a miss to shared data incurs at least one request and response message pair.

All consistency information in lazy-release-consistency systems is piggybacked on synchronization messages (barrier messages in the case of compiler-parallelized applications). By contrast, diff requests are inherently two-way, and so cost two messages. On the minus size, if sharing patterns are not stable, out-of-date copysets will cause data to be sent to processors that do not need it. Correctness is not affected, but the unneeded flushes cause unnecessary overhead.

To improve the effectiveness of the hybrid protocol, we enhanced it in two ways for CVM. First, we flush updates for data at barrier synchronization points to enable data to be piggybacked on synchronization messages (where possible) and multiple updates to be aggregated in a single message. Second, we provide a flexible user-level (i.e., non-kernel) interface for specifying the coherence for a page or range of pages. This flexibility is important because applications typically have phase shifts when data access patterns change. CVM allows 1) dynamically changing the coherence type of a page to either invalidate or update, 2) clearing the copyset of a page, 3) adding or removing processors from the copyset of a page.

# 4 Experimental Results

This sections presents our experimental results. We discuss our experimental environment, present our overall results, discuss the effect of two compiler-directed optimizations, and then summarize our results.

## 4.1 Experimental Environment

Our experimental environment consists of a 16-node IBM SP2, although all performance numbers reflect eight-processor executions. The SP-2 has a high-performance Omega switch in which each bi-directional link is capable of a sustained bandwidth of approximately forty megabytes per second. Each processor is a 66MHz RS/6000 Power2 processor.

Our system is based on unreliable UDP sockets, communicating over the switch. Simple RPCs take 160 $\mu$sec, and eight-processor barriers take a minimum of 669 $\mu$secs. Misses on shared data take a minimum of 939 $\mu$secs, including both system time and the cost of retrieving a 4096-byte page across the switch. Misses are detected by changing page protections and specifying handlers to be called on an inappropriate access. The operating system overhead of such a handler call is 128 $\mu$secs. Operating system overhead for calling handlers for incoming messages is similar.

### 4.2 Applications

We evaluated the performance of our prototype compiler/software DSM interface with eight kernels and program shown in Table 1. Except for `mult`, applications are composed of stencil computations and reductions common in dense-matrix scientific codes. `dot`, `erle`, `jacobi`, and `swm` contain reductions. `erle` and `swm` are small programs containing hundreds of lines. The remaining applications are kernels, several taken from the Livermore Loops.

| Name | Description | Problem Size |
|---|---|---|
| adi | ADI Fragment (Livermore 8) | 32K |
| dot | Dot Product (Livermore 3) | 512K |
| erle | Erlebacher (3D Tridiagonal Solver) | $96^3$ |
| expl | Explicit Hydrodynamics (Livermore 18) | $512^2$ |
| jacobi | Jacobi Iteration w/Convergence Test | $512^2$ |
| mult | Matrix Multiply | $300^2$ |
| rb | Red-Black Successive-Over-Relaxation | $1K^2$ |
| swm | Shallow Water Model | $512^2$ |

**Table 1** Applications

These applications are written in Fortran and typically contain an initialization section followed by tens or hundreds of iterations of a time step loop. They were automatically parallelized by the Stanford SUIF parallelizing compiler, with close to 100% of the computation in parallel regions. A simple block scheduling policy assigns contiguous iterations of equal or near-equal size to each processor, resulting in a consistent computation partition that encourages good locality. The resulting C output code was compiled by `g++` version 2.6.3 with the -O2 flag, then linked with the SUIF run-time system and the CVM libraries to produce executable code on the IBM SP-2.

### 4.3 Speedup and Execution Time Breakdown

Figure 1 shows speedup for our eight applications. Our speedup graph presents the speedup of the best combination of the two optimizations. The performance of our applications covers a broad range. Unsurprisingly because of its high computation-to-communication ratio, `mult` gets a speedup of over seven. `Dot`, `jacobi`, and `expl` get speedups of around six, while `Adi` and `rb` achieve medium speedups. The two programs, `swm` and `erle`, exhibit very little speedup at all.

The causes of the poor performance in some applications can be seen in Figure 2. This chart breaks down execution time into six categories: application processing time, time spent in communication routines, miss handling time, time spent "flushing" data in our update protocol, garbage collection time, and time spent waiting at barriers. This latter time is almost entirely load imbalance. While the compiler-generated code is perfectly balanced, time spent handling faults, diff requests, and flush messages delays processors unequally between barriers.

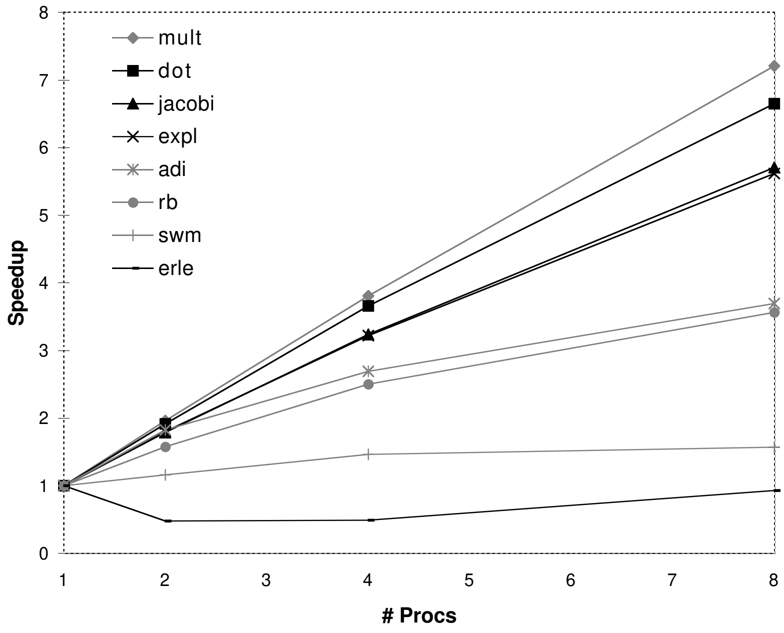


**Figure 1** 8-Proc Speedup

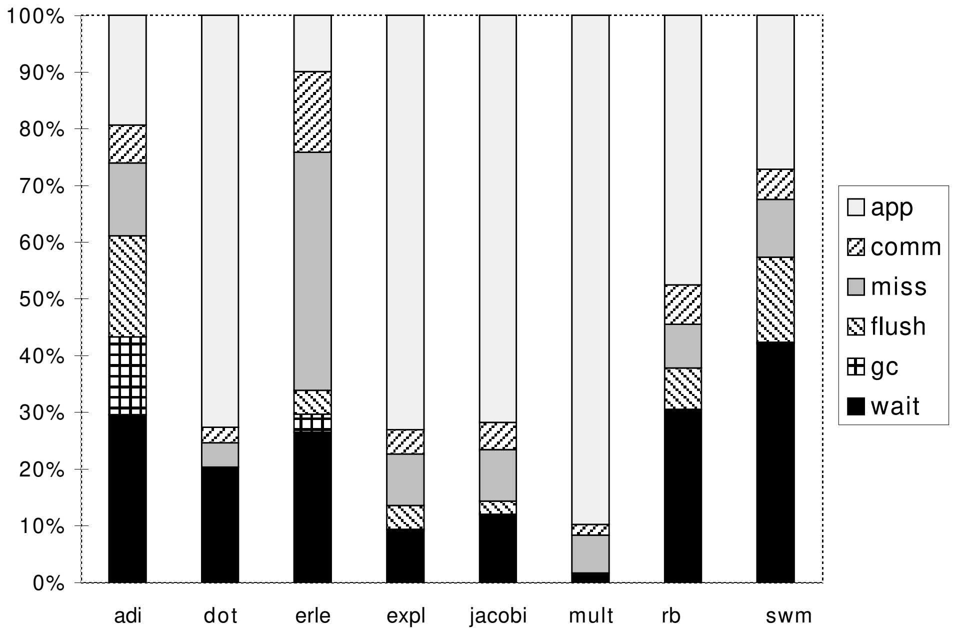


**Figure 2** Breakdown of Execution Time

"Miss" time includes system time spent calling the fault handler and changing page protections, as well as all remote requests needed in order to validate the page. This category is deceptively small, since variation in miss handling time among processors appears to be the primary cause of load imbalance. Hence, any reduction of miss handling time is likely to reduce barrier wait time as well. Garbage collection time is minimal for all applications except `adi`, and is only large for `adi` only because the breakdown reflects an execution that uses hybrid updates, which increase the number of diff creations by a factor of eight. Garbage collection does not occur at all for `adi` when hybrid updates are turned off.

### 4.4 Evaluating Hybrid Update Protocol

Recall that when enabled, our compiler automatically insert calls to DSM routines that mark address ranges to be kept coherent using a hybrid update protocol; updates are *flushed* at barriers. For a given page, locally modifications are flushed to processors named by a page's local copyset prior to each barrier. For applications with non-adaptive reference patterns, such as those in our test suite, copyset information accurately reflects stable sharing patterns by the second iteration. The current algorithm used to select data is fairly imprecise, and marks all arrays accessed in parallel as data requiring updates.

Table 2 contains statistics on diffs, page invalidations, remote misses, and messages both with and without compiler-generated barrier flushes. Because of the interference with lazy diffing, described below, barrier flushes uniformly create more diffs. However, the difference is minor in five of the programs, indicating they have stable one-to-one sharing patterns. In all cases, barrier flushes reduce the number of page invalidations and remote misses.

If sharing patterns are not stable, out-of-date copysets will cause data to be sent to processors that do not need it. Correctness is not affected, but the unneeded flushes cause unnecessary overhead. The "Percent Used" column shows that such is the case for `erle` and `adi`, the two applications for which barrier flushes are not helpful. The problem seems to be the compiler neglected to clear the copysets of shared variables after the initialization phase, causing extra updates in the actual time step loop.

The `adi` application also suffers from a less obvious disadvantage of barrier flushes that occurs when data is consumed less frequently than it is modified. For example, consider a three-barrier application executing on processors $p$ and $q$. Processor $p$ modifies page $i$ during each of the first two barrier epochs, and $q$ reads page $i$ in the third. Multi-writer DSMs such as CVM typically use a *lazy diffing* diffing scheme, which means that they delay actually creating a diff until it is requested. In the above case, without barrier flushes, the lazy scheme would not create a diff until $q$ requests the modified data from $p$ in the third epoch. Hence, only one diff for page $i$ is created during each iteration. With barrier flushing enabled, diffs are created and flushed in each of the first two epochs, resulting in twice as many diffs being created overall.

### 4.5 Evaluating Customized Reductions

Table 3 contains statistics describing executions with and without customized reduction support for the four applications that use reductions. Without customized reductions, accumulations occur through mutually exclusive updates to shared memory. With customized reduction support, reduction records created and piggybacked on barrier synchronization messages. The results show customizing reductions is quite effective for reducing access misses for the reduction. The effect on performance is dependent on the frequency reductions are executed. For `dot` and `jacobi` customized reductions are important since reductions are frequent, while for `erle` and `swm` the effect is less (though still noticeable) because reductions are rare.

### 4.6 Communication Requirements

Table 4 lists message and bandwidth totals for the applications with both optimizations turned on. "Page" requests are only sent on the initial access to a page, or the first access to a page after a garbage collection. "Diff" requests are used thereafter to bring a page up to date. The message total reflects the fact that

| | Invals | | Misses | | Diffs | | Msgs | | Percent | Speedup |
|---|---|---|---|---|---|---|---|---|---|---|
| | w/o | w/ | w/o | w/ | w/o | w/ | w/o | w/ | Useful | With |
| adi | 12424 | 4122 | 13285 | 5032 | 11628 | 95590 | 35260 | 102709 | 70 | -58% |
| dot | 1808 | 1794 | 1815 | 1801 | 3 | 3 | 31712 | 31698 | 100 | +0% |
| erle | 7810 | 5010 | 7547 | 6027 | 4306 | 4417 | 21650 | 19187 | 83 | -14% |
| expl | 12628 | 4150 | 12697 | 4279 | 7558 | 8127 | 30644 | 17598 | 96 | +8% |
| jacobi | 9890 | 3616 | 9897 | 3651 | 4502 | 4530 | 28328 | 20023 | 100 | +7% |
| mult | 1670 | 1318 | 5101 | 4764 | 861 | 872 | 16922 | 16605 | 100 | +1% |
| rb | 13015 | 1836 | 13036 | 1871 | 6430 | 12030 | 48602 | 37507 | 100 | +10% |
| swm | 71267 | 4307 | 71436 | 4595 | 50505 | 53026 | 300906 | 74250 | 100 | +41% |

**Table 2** Hybrid Protocol

| | Invals | | Misses | | Diffs | | Msgs | | Speedup |
|---|---|---|---|---|---|---|---|---|---|
| | w/o | w/ | w/o | w/ | w/o | w/ | w/o | w/ | With |
| dot | 8463 | 1794 | 8477 | 1801 | 9004 | 3 | 134515 | 31698 | +56% |
| erle | 6061 | 5010 | 6856 | 6027 | 4553 | 4417 | 22463 | 19187 | +8% |
| jacobi | 5716 | 3616 | 3977 | 3651 | 5220 | 4530 | 30489 | 20023 | +30% |
| swm | 4310 | 4307 | 4600 | 4595 | 53034 | 53026 | 74353 | 74250 | +1% |

**Table 3** Reduction Support

all messages except barrier flushes require a response. The data shows a large amount of data is being communicated, perhaps more than expected.

### 4.7 Page Alignment and Multiple Writers

The numbers in this paper reflect a multi-writer protocol. CVM also supports several other protocols, including a single-writer protocol [8]. Multiple-writer protocols have several advantages. They alleviate the "ping-pong" effect when applications exhibit substantial write sharing (whether it be true sharing or false sharing). They allow processors to make a local decision to modify a page, i.e. no network communication is needed in order to modify a page that is already valid, but in a read-only state. Multiple writer protocols are also generally more stable in the face of different synchronization and sharing patterns.

Table 5 shows diffs created, misses and speedup for the eight applications. In all cases, barrier flushes have been turned off for the multi-writer protocol because they have not been implemented in the single-writer protocol. As table 5 shows, most of the programs in this study are regular and have coarse-grained sharing. The two protocols therefore perform similarly. Two exceptions, `adi` and `swm`, perform significantly worse under a single-writer protcol than under multiple-writer protocols, primarily because of fine-grained, dynamic sharing.

### 4.8 Discussion

Our experimental results demonstrate that compiler-generated code can perform well on DSM systems, *provided* that they have sufficient granularity of parallelism and are able to provide hints to DSM system as to how data is being used. Even with our optimized compiler/software DSM interface, performance is poorer than expected for programs such as `erle` and `swm` that do not have sufficiently coarse-grain parallelism. We

| Programs | Messages | | | | | Bandwidth (kbytes) |
|---|---|---|---|---|---|---|
| | Barrier | Page | Flush | Diff | Total | |
| adi | 2889 | 3377 | 78567 | 5805 | 102709 | 2890 |
| dot | 14035 | 1800 | 14 | 7 | 31698 | 9285 |
| erle | 585 | 4665 | 1845 | 3421 | 19187 | 36420 |
| expl | 2555 | 4166 | 3790 | 183 | 17598 | 1163 |
| jacobi | 4235 | 3615 | 4187 | 68 | 20023 | 33183 |
| mult | 315 | 4299 | 357 | 3510 | 16605 | 32775 |
| rb | 11263 | 1824 | 11179 | 77 | 37507 | 44459 |
| swm | 12719 | 3409 | 38310 | 1842 | 74250 | 112456 |

**Table 4** Communication Requirements

| | Misses | | Msgs | | Speedup | |
|---|---|---|---|---|---|---|
| | Single | Mult | Single | Mult | Single | Mult |
| adi | 22073 | 13271 | 78990 | 35247 | 2.48 | 3.67 |
| dot | 1815 | 1815 | 31714 | 31712 | 6.65 | 6.65 |
| erle | 9396 | 6027 | 24035 | 21650 | 0.94 | 0.95 |
| expl | 14590 | 12697 | 40812 | 30658 | 5.23 | 5.10 |
| jacobi | 9903 | 9897 | 28350 | 28328 | 5.65 | 5.11 |
| mult | 5246 | 5101 | 11812 | 16922 | 3.94 | 3.82 |
| rb | 13050 | 13036 | 48602 | 48602 | 3.10 | 3.18 |
| swm | 94001 | 71436 | 288458 | 74250 | 0.64 | 1.11 |

**Table 5** Single vs. Multiple Concurrent Writers

need to make the interface more efficient so that we can achieve speedups with smaller granularities of parallelism. Significant improvements are obviously needed before software DSMs provide a sufficiently efficient platform for parallelizing compilers. Based on our experiences, we point out some likely avenues for explorations in the next section.

## 5 Additional Improvements

We believe additional improvements are needed in both the compiler and software DSM to make the combination efficient. We begin by discussing improvements to the compiler. *Note: for the final version of the paper we expect to have several of these enhancements implemented and experimentally evaluated.*

### 5.1 Better Update Classification

First, we anticipate doing a much better job on providing annotations for variables to guide flushing updates at barriers. Currently, our compiler analysis is imprecise and does little better than mark entire shared regions as "update", and the compiler currently does not mark phase changes in the programs. We plan to extend the compiler so it can differentiate between data that is accessed with stable sharing patterns, and data that whose sharing pattern changes dynamically. The first category is appropriate for "update" annotations; the second is not.

Many applications go through different stages, where sharing patterns are stable within stages but change between them. For example, the `erle` application has two different phases. Since our compiler does not yet detect phase changes, copysets become poor predictors of future accesses in the second phase. Hence the low "Percent Useful" number in Table 2. Merely detecting such changes and directing the DSM to clear copysets would eliminate most problems associated with phase changes. Creating new sharing annotations to add processors to copysets at this point would eliminated almost all of the rest. We expect this type of support to be even more essential for larger and more complex applications.

### 5.2 Improving Memory Layout

Message-passing programs have good spatial locality, since data assigned to each processor is placed in contiguous memory locations. The same may not be true for shared-memory programs, since the data assigned to each program may be scattered through the address space depending on how it has been partitioned. Poor spatial locality for data can cause false sharing in single-writer protocols and increase diff creation in multi-writer protocols. To improve spatial locality of local data, the compiler may decide to reindex array references to make local sections of each array contiguous. However, scalar optimizations are required to clean up modulo and division operations inserted into array subscripts [3]. Since the compiler is already building a structure for all shared variables, it should also attempt to page align shared data to improve spatial locality at the page level.

### 5.3 Packing Nonlocal Data

Software DSM systems may waste significant communication bandwidth for nonlocal data accesses with poor spatial locality. Extra swapping of pages to disk may occur if the number of pages exceed available memory. Message-passing programs avoid these problems by combining nonlocal accesses in a single message to reduce communication costs. Shared-memory compilers can obtain also benefit by copying nonlocal data with poor spatial locality into contiguous buffers. The compiler must first apply communication analysis to detect nonlocal accesses. If the nonlocal data is not contiguous, then the compiler must insert code to copy the data to contiguous buffers (one for each processor). The placement copy code can be determined by data dependences using message vectorization [2, 19]. The compiler must also modify the code so data so nonlocal accesses are made to the buffers.

### 5.4 Message Library Support

Figure 2 shows that a large amount of time is spent performing system-related chores, even in these relatively simple applications. Part of the problem is the underlying communication mechanism. The numbers in this paper reflect using UDP sockets as a communication substrate. Sockets are very inefficient; round-trip latency is usually thousands of cycles, even on a system with a fast network, such as the SP-2. CVM also runs on top of IBM's implementation of MPI, which has much lower latencies and supports large message sizes. Even with the basic performance advantages, however, MPI-based DSM usually performs less well than UDP-based DSM. The reason is that MPI (as well as the current draft of MPI2) provides no support for asynchronous invocation of handlers upon message receipt. These handlers are necessary for timely handling of diff, lock, and page requests. The currently use the standard solution, i.e. relying on polling whenever messages are sent. By using the compiler to automatically insert probes into application code, we should be able to obtain consistent performance from MPI.

### 5.5 Retargeting DSM Support

Synchronization usage of automatically-parallelized scientific codes often clashes with the synchronization model assumed by DSMs. Many DSMs support a broader range of synchronization models than needed for most automatically parallelized scientific code. Synchronization in scientific codes consists primarily of barriers and reductions, i.e. global operations. DSMs usually target end users directly, so they support many different synchronization types, including local synchronization such as locks and condition variables.

Such support has a price, much of the consistency-related machinery in systems such as CVM is dead weight when running scientific codes. We are working on a pared-down version of CVM that is specifically tailored to support barriers, reductions, and limited producer-consumer interaction.

### 5.6 Reduction Support

We currently support reductions by centralizing all reduction operations at the master processor, which can cause a bottleneck for applications with large amounts of reductions. Reductions in such systems can be distributed on a per-page, or per-reduction variable basis. For example, we could require all reductions to be performed at the processor that owns the page that contains the reduction variable. A disadvantage of the distributed approach is that it requires additional messages. In the centralized approach, all reduction traffic is piggybacked on existing barrier messages. More experiments will be needed.

## 6 Related Work

While there has been a large amount of research on software DSMs [9, 11, 29], we are aware of only a few projects combining compilers and software DSMs. Bershad et al. [5] maintain coherence by using a compiler to update a software dirty bit on shared-memory accesses. CVM, like most software DSMs, relies on the virtual memory system to detect shared memory updates. Results show that the software communication overhead usually dominates the memory management overhead.

Mukherjee *et al.* compared the performance of explicit message-passing programs with shared-memory programs [28] on Typhoon, a Flexible-Shared-Memory machine implemented on top of a CM-5 [30, 31]. Results show that with suitable extensions to the coherence protocol, the shared-memory program was able to match the performance of the optimized message-passing program utilizing Chaos [10]. The authors point out that a compiler like SUIF can take advantage of the extensible coherence protocol to improve performance.

The SUIF compiler draws on a large body of work on techniques for identifying parallelism [6, 17, 35]. Previous researchers have examined shared-memory compilation issues such as improving locality [26] and reducing false sharing [7, 12, 34], but their techniques were mostly needed for single-writer hardware coherence protocols. Granston and Wishoff suggest a number of compiler optimizations for software DSMs [15]. These include tiling loop iterations so computation is on partitioned matching page boundaries, aligning arrays to pages, and inserting hints to use weak coherence. No implementation or experiments are provided. CVM uses a multi-writer release consistency protocol, so these optimizations are not as vital as for a sequentially-consistent single-writer protocol.

Mirchandaney *et al.* described the design of a compiler for Treadmarks, a software DSM [27]. They propose *section locks* and *broadcast barriers* to guide eager updates of data, integrating *send*, *recv* and *broadcast* operations with the software DSM, and reductions based on multiple-writer protocols. Their proposal is similar to portions of our SUIF/CVM interface, but we have improved upon their approach based on experiences with an existing shared-memory compiler implementation.

## 7 Conclusions

Current parallelizing compilers for message-passing machines only support a limited class of data-parallel applications. In this paper we investigate whether we can eliminating this restriction by combining a powerful shared-memory parallelizing compiler with an advanced software DSM system. Preliminary results show simply combining the parallelizer and software DSM yields very poor performance. The compiler/software DSM interface can be improved based on relatively little compiler input by: 1) combining synchronization and parallelism information communication on parallel task invocation, 2) employing customized routines for evaluating reduction operations, and 3) selecting a hybrid protocol to presend data by flushing updates at barriers. Though these optimizations yield decent speedups for program kernels with coarse-grain parallelism, performance for programs with smaller granularity of parallelism is still poor. Communication overheads in software DSMs are quite high, leaving little room for sloppy compilation. Based on our experiences, we believe additional compiler analysis and optimization and further software DSM improvements are necessary to achieve good performance for entire applications.

## References


[1] A. Agarwal, R. Bianchini, D. Chiaken, K. Johnson, D. Kratz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife machine: Architecture and performance. In *Proceedings of the 22th International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, June 1995.

[2] S. Amarasinghe and M. Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, Albuquerque, NM, June 1993.

[3] J. Anderson, S. Amarasinghe, and M. Lam. Data and computation transformation for multiprocessors. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

[4] J. Bennett, J. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Seattle, WA, March 1990.

[5] B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon. The Midway distributed shared memory system. In *Proceedings of the '93 CompCon Conference*, pages 528–537, February 1993.

[6] W. Blume et al. Polaris: The next generation in parallelizing compilers,. In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, Ithaca, NY, August 1994.

[7] W. Bolosky and M. Scott. False sharing and its effect on shared memory performance. In *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, San Diego, CA, September 1993.

[8] R. Bryant, P. Carini, H.-Y. Chang, and B. Rosenburg. Supporting structured shared virtual memory under Mach. In *Proceedings of the 2nd Mach Usenix Symposium*, November 1991.

[9] 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.

[10] R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–479, September 1994.

[11] 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.

[12] S. J. Eggers and T. E. Jeremiassen. Eliminating false sharing. In *Proceedings of the 1991 International Conference on Parallel Processing*, St. Charles, IL, August 1991.

[13] B. Falsafi, A. Lebeck, S. Reinhardt, I. Schoinas, M. Hill, J. Larus, A. Rogers, and D. Wood. Application-specific protocols for user-level shared memory. In *Proceedings of Supercomputing '94*, Washington, DC, November 1994.

[14] 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.

[15] E. Granston and H. Wishoff. Managing pages in shared virtual memory systems: Getting the compiler into the game. In *Proceedings of the 1993 ACM International Conference on Supercomputing*, Tokyo, Japan, July 1993.

[16] M. Hall, S. Amarasinghe, B. Murphy, S. Liao, and M. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995.

[17] M. W. Hall, S. Amarasinghe, and B. Murphy. Interprocedural analysis for parallelization: Design and experience. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, San Francisco, CA, February 1995.

[18] M. Heinrich, J. Kuskin, D. Ofelt, J. Heinlein, J.-P. Singh, R. Simoni, K. Gharachorloo, J. Baxter, D. Nakahira,

M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. 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 (ASPLOS-VI)*, San Jose, CA, October 1994.

[19] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[20] P. Keleher. Multiple writers considered harmful. Technical Report CS-TR-3543, Dept. of Computer Science, University of Maryland at College Park, October 1995.

[21] 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.

[22] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook*. The MIT Press, Cambridge, MA, 1994.

[23] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simon, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In *Proceedings of the 21th International Symposium on Computer Architecture*, April 1994.

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

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

[26] E. Markatos and T. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):379–400, April 1994.

[27] R. Mirchandaney, S. Hiranandani, and A. Sethi. Improving the performance of DSM systems via compiler involvement. In *Proceedings of Supercomputing '94*, Washington, DC, November 1994.

[28] S. Mukherjee, S. Sharma, M. Hill, J. Larus, A. Rogers, and J. Saltz. Efficient support for irregular applications on distributed-memory machines. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

[29] B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE Computer*, 24(8):52–60, August 1991.

[30] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-level shared memory. In *Proceedings of the 21th International Symposium on Computer Architecture*, April 1994.

[31] I. Schoinas, B. Falsafi, A. Lebeck, S. Reinhardt, J. Larus, and D. Wood. Fine-grain access control for distributed shared memory. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, San Jose, CA, October 1994.

[32] J.P. Singh, T. Joe, A. Gupta, and J. Hennessy. An empirical comparison of the Kendall Square Research KSR-1 and Stanford DASH multiprocessors. In *Proceedings of Supercomputing '93*, Portland, OR, November 1993.

[33] E. Su, A. Lain, S. Ramaswamy, D. J. Palermo, E. W. Hodges IV, and P. Banerjee. Advanced compilation techniques in the PARADIGM compiler for distributed-memory multicomputers. In *Proceedings of the 1995 ACM International Conference on Supercomputing*, Barcelona, Spain, July 1995.

[34] J. Torrellas, M. Lam, and J. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, June 1994.

[35] P. Tu and D. Padua. Automatic array privatization. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[36] R. Wilson et al. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 1994.