Enhancing Software DSM for Compiler-Parallelized Applications

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. We demonstrate such a system by combining the SUIF parallelizing compiler and the CVM software DSM. Innovations of the system include compiler-directed techniques that: 1) combine synchronization and parallelism information communication on parallel task invocation, 2) employ customized routines for evaluating reduction operations, and 3) select a hybrid update protocol that pre-sends data by flushing updates at barriers. For applications with sufficient granularity of parallelism, these optimizations yield very good speedups eight processors on an IBM SP-2 and DEC Alpha cluster, usually matching or exceeding the speedup of equivalent HPF and message-passing versions of each program. Based on our experimental results, we point out areas where additional compiler analysis and software DSM improvements can be used to achieve good performance on a broader range of applications.

1 Introduction

Increasingly powerful processor and network architectures make so-called “meta-computers” (loosely-coupled computers communicating via messages) a tempting platform on which to run large parallel and distributed applications. Unfortunately, writing efficient message-passing programs is difficult, error-prone, and tedious, and data-parallel languages such as High Performance Fortran (HPF) [18] may prove overly restrictive. We believe that the combination of shared-memory parallelizing compilers and sophisticated runtime systems presents one of the most promising approaches towards addressing this key problem.

This paper presents our experience using the CVM [15] software distributed shared memory (DSM) system as a compilation target for the SUIF [12] shared-memory compiler. SUIF automatically parallelizes sequential applications and allows users to benefit from sophisticated program analysis. The use of CVM as a compilation target hides the details of the underlying message-passing architecture and allows the compiler-generated code to assume shared memory semantics.

By combining these two technologies, we create a programming environment that is flexible and easy to use, since scientists are no longer required to write message passing programs or use 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. This combination has the advantage of producing programs that can run on the large-scale parallel machines as well as the low-end, but more pervasive multiprocessor 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.

By studying the performance of compiler-parallelized programs on CVM, we are also helping to validate the efficiency of software DSMs in general. Previous studies have relied on carefully hand-tuned parallel programs such as the Splash Benchmarks. By achieving good performance for compiler-parallelized applications, which are much less tuned for the underlying memory system, we show that software DSMs are efficient enough to support a wider class of applications than previously demonstrated.

This research was supported by NSF CAREER Development Awards #CCR9624803 in Operating Systems and #ASC9625531 in New Technologies.
The IBM SP-2 and DEC Alpha Cluster were provided by NSF CISE Institutional Infrastructure Award #CDA9401151 and grants from IBM and DEC.

1.1 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 makes a number of contributions towards answering this question:

- experimental evaluation of an actual compiler/DSM system on two machine architectures
- DSM enhancements to
  1. combine synchronization and application data messages with parallel task invocation
  2. eliminate synchronization and piggyback messages for reduction operations
  3. selectively use an update flush protocol for dynamically shared data
- comparison with data-parallel (HPF) and message-passing (MPI) versions of each program

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, followed by experimental results. We conclude with a discussion of potential improvements to our system and related work.

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. 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.

Shared-memory compilers enjoy a significant advantage over HPF compilers because they do not need precise information on *all* interprocessor communication. Because of this generality, current shared-memory compilers can efficiently support a much larger set of applications than current HPF compilers. In this paper we show that for many applications, slightly extending analysis in a shared-memory compiler (for data likely to be communicated to other processors) can yield comparable performance to full-blown communication analysis in HPF compilers, with much greater flexibility and less effort.

2.2 CVM

The DSM target used in this work is CVM, a software DSM that supports multiple protocols and consistency models. Like commercially available systems such as TreadMarks [17], CVM is written entirely as a user-level library and runs on most UNIX-like systems. Unlike TreadMarks, CVM was created specifically as a platform for protocol experimentation.

The system is written in C++, and opaque interfaces are strictly enforced between different functional units of the system whenever possible. The base system provides a set of classes that implement a generic protocol, lightweight threads, and network communication. The latter functionality consists of efficient, end-to-end protocols built on top of UDP.

New shared memory protocols are created by deriving classes from the base `Page` and `Protocol` classes. Only those methods that differ from the base class's methods need to be defined in the derived class. The underlying system calls protocol hooks before and after page faults, synchronization, and I/O events take place. Since many of the methods are inlined, the resulting system is able to perform within a few percent of TreadMarks, a severely optimized system, running a similar protocol. However, CVM was designed to take advantage of generalized synchronization interfaces, as well as to use multi-threading for latency toleration. We therefore expect the performance of the fully functional system to improve over the existing base. In order to simplify the comparison process, however, we do not use either of these techniques in this study.

Memory Consistency - CVM's primary protocol implements a multiple-writer version of lazy release consistency [16], which is a derivation of *release consistency* (RC) [9]. Release consistency a processor to delay making modifications to shared data visible to other processors until special *acquire* or *release* synchronization accesses occur. The propagation of 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 [9]. In practice, most shared-memory programs require little or no modifications to meet these requirements.

Lazy release consistency (LRC) allows the propagation of modifications to be further postponed until the time of the next subsequent acquire of a released synchronization variable. At this time, the acquiring processor determines which modifications it needs to see according to the definition of LRC. 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. At an acquire, processor p sends its current vector timestamp to the previous releaser of the same synchronization variable, 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 LRC [7].

False Sharing - *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 [3] such as that implemented by CVM avoid false sharing by allowing two or more processors to simultaneously modify local copies of the same shared page.

These concurrent modifications are merged using *diffs* to summarize the updates. A diff is created by performing a page-length comparison between the current contents of the page and a *twin* of the page that was created at the first write access. If each concurrent writer summarizes its modifications as a diff, the system can create a copy that reflects all modifications by applying the concurrent diffs to the same copy. Concurrent diffs only overlap if the same location is written by multiple processors without intervening synchronization, which is probably a data race.

Access misses - CVM uses 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 the page's owner. 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

Our system consists of the Stanford SUIF parallelizing compiler [12] and the CVM software DSM system [15]. A simple interface was produced by porting the SUIF run-time system to the CVM API. Because shared data in CVM must be global, we also implemented passes in the compiler to promote all shared local variables to globals, and to pack all shared global variables into a single contiguous global structure.

3.1 Optimizations

The simple interface presented for SUIF and CVM produces a working system, but contains many inefficiencies, some of which may be eliminated with enhancements to the software DSM that rely on lightweight 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, producer p receives a large number of requests, adding a serial bottleneck. Further, if X overlaps more than one page, the pages are retrieved serially as they are accessed.

3.1.1 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 shared memory reserved for the compiler run-time system. Those variables are then automatically updated with new values with the synchronization messages for the barrier.

3.1.2 Customized Reductions

Another opportunity for improving the compiler/software DSM interface is in customized support for reductions. 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. A straightforward implementation would use ordinary accesses to shared memory, guarded by lock variables in order 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 supports reductions by copying the reduction operator and local reduction data into a local reduction record. All such records are appended to the next outgoing barrier arrival message. The master thread then performs all 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.

3.1.3 Flush 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 *copyset* information collected by the underlying software DSM system.

CVM track copies of shared pages by using copysets, which are bitmaps that specify which processors cache a given 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 *flushed* 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. Under the flush protocol, access misses can be then be eliminated by updating processors on the copyset for each page, sending the data before it is accessed.

We modified the compiler to automatically insert calls to DSM routines to mark pages to be *flushed* at barriers. For a given page, local modifications are then 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.

Compiler analysis needed to use such a protocol is much simpler than communication analysis needed in HPF compilers. The identities of the sending/receiving processors do not need to be computed at compile time, and the compiler does not need to be 100% correct since the only effect is on efficiency, not correctness. Instead, the compiler only needs to locate data that will likely be communicated in a stable pattern, then insert calls to DSM routines to apply the flush protocol for those pages at the appropriate time. More precise compiler analysis can be used to explicitly clear or set the copysets of data to be communicated.

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 and page faults. A second advantage is that lost flush messages do not affect correctness, only performance. Flush messages do not have to be 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.

The basic flush protocol as described above was modified in two ways for this study. 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 section 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

We evaluated our optimizations on a cluster of DEC Alpha workstations and an IBM SP-2. Our DEC Alpha cluster consists of eight DEC Sables multiprocessors with four 250MHz Alpha 20064 processors and 256 megabytes of memory each, operated under DEC Unix version 3.2D. The nodes are connected by a 155-MBit/sec ATM switch. Results use only a single processor per node. Using more than one processor per node resulted in up to 15% performance degradation relative to a system containing the same number of processors, but only a single processor per node. This degradation arises primarily from contention at the network interface, and has been documented on similar systems elsewhere [8, 25].

On the DEC cluster, CVM processes communicate via unreliable UDP sockets over the ATM switch. Simple RPCs take 160 μsec, and eight-processor barriers take a minimum of 1836 μsecs. Misses on shared data take a minimum of 1388 μsecs, including both system time and the cost of retrieving a 8192-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 μsecs. Operating system overhead for calling handlers for incoming messages is similar.

We also present results from an IBM SP-2 with 66MHz RS/6000 Power2 processors operating AIX 4.1 connected by a 120 Mbit/sec bi-directional Omega switch. Simple RPCs on the SP-2 require 160 μsecs. A *one-hop* page miss, where the page manager is also the owner, requires two messages and 939 μsecs. *Two-hop* page misses require three messages and 1376 μsecs. In the best case, AIX requires 128 μsecs to call user-level handlers for page faults, and `mprotect` system calls require 12 μsecs. However, virtual memory primitive costs in the current system are location-dependent, occasionally increasing these costs to a millisecond or more.

4.2 Applications

We evaluated the performance of our compiler/software DSM interface with the nine programs shown in Table 1. ADI, EXPL, and RB are dense stencil kernels typically found in iterative PDE solvers. JACOBI is a stencil kernel combined with a convergence test that checks the residual value using a `max` reduction. DOT calculates the inner product of two vectors using a `sum` reduction.

IRREG models an iterative PDE solver on a randomly generated irregular mesh. MULT performs matrix multiplication. SWM and TOMCATV are programs from the SPEC benchmark suite containing a mixture of stencils and reductions. The arrays in TOMCATV have been transposed to improve data locality.

Name	Description	Problem Sizes		Granularity (secs)	
		Small	Large	Small	Large
adi	ADI Fragment (Livermore 8)	32K	64K	0.31	0.63
dot	Dot Product (Livermore 3)	256K	512K	0.10	0.28
expl	Explicit Hydrodynamics (Livermore 18)	256^2	512^2	0.06	0.34
irreg	Irregular Solve Over Mesh	500K	1000K	0.06	0.12
jacobi	Jacobi Iteration w/Convergence Test	512^2	1024^2	0.06	0.91
mult	Matrix Multiply	300^2	400^2	1.83	4.33
rb	Red-Black Successive-Over-Relax.	512^2	1024^2	0.01	0.14
swm	Shallow Water Model (SPEC)	512^2	750^2	0.10	0.20
tomcatv	Vector Mesh Generation (SPEC)	256^2	512^2	0.04	0.15

Table 1 Applications

In Table 1, the "Granularity" column refers to the average length in seconds of a parallelized loop. Except where indicated, numbers below refer to the larger data set for each application. All applications were originally written in Fortran, and typically contain an initialization section followed by iterations of a time step loop. Statistics and timings are collected after the initialization section.

4.3 Programming Models

In our experiments, CVM applications written in Fortran 77 were automatically parallelized by the Stanford SUIF parallelizing compiler version 1.0, 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 DEC Alphas and IBM SP-2.

We also evaluated the efficiency of our CVM shared-memory interface by comparing its performance against data-parallel (HPF) and message-passing (MPI) versions of each program. High Performance Fortran (HPF) applications were created by manually translating from Fortran to Fortran 90, with HPF data decompositions added for each array. On the IBM SP-2 we used the IBM HPF compiler [11] with the -O2 flag. On the DEC cluster we used the DEC HPF compiler `f90` version 2.0-1 with the -O2 -wsf -fast flags.

Message-passing versions of each program were created using calls to communication routines specified under Message Passing Interface (MPI). On the IBM SP-2 we used the MPL version 2 implementation of MPI; on the DEC Alpha cluster we used the MPICH version 1.0.12 implementation of MPI. MPI versions of ADI, DOT, EXPL, and JACOBI were generated using the Fortran D compiler [13]. Previous experiments show the resulting programs achieve performance close to optimized hand-written message-passing programs [14]. MPI versions of MULT and RB were created by hand. These programs represent message-passing programs written with a reasonable amount of effort, not programs highly customized for performance.

Figures 1 and 2 show CVM, HPF, and MPI speedups for both large and small data sets for each of our applications on the IBM SP-2 and DEC Alpha cluster, respectively. Speedup is calculated relative to the sequential versions of each program, without any calls to the parallel run-time system. Although we were careful to ensure that paging does not occur in the single-processor runs of any applications, cache effects are enough to cause superlinear speedup in some cases. Sequential execution times for CVM and HPF programs differ slightly, since HPF programs have been rewritten in Fortran 90, but are generally comparable.

The performance of our applications cover a broad range. As expected, both systems perform better with larger data sets. CVM speedups are quite good. For the larger data sets over eight processors, CVM has an average speedup of 7.2 on the DEC Alpha cluster and 5.0 on the IBM SP-2. These results show that shared-memory compilers targeting an enhanced software DSM can achieve excellent results on message-passing systems for a moderate number of processors, at least for applications with sufficient granularity of parallelism.

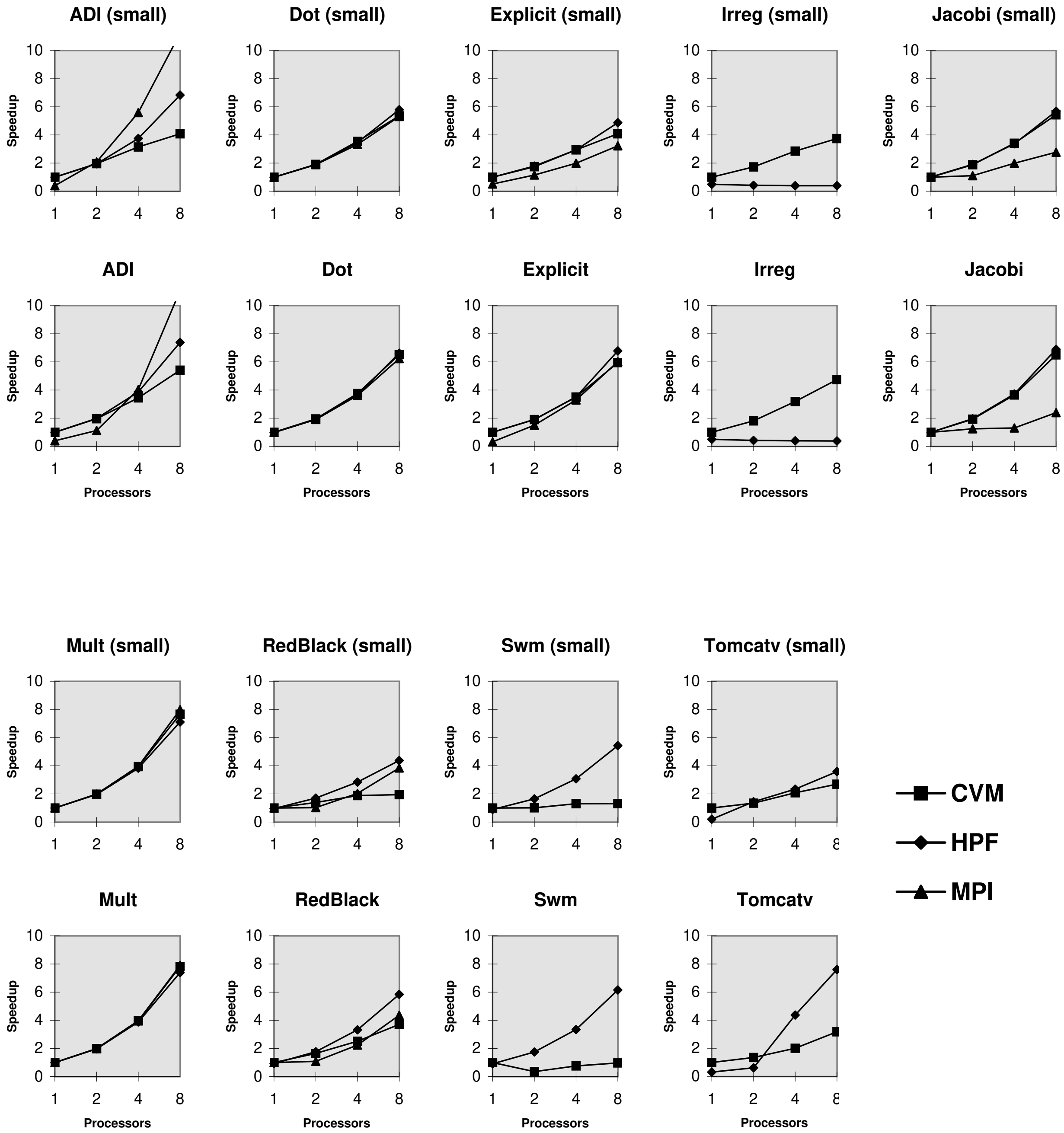

Figure 1 Speedups for IBM SP-2

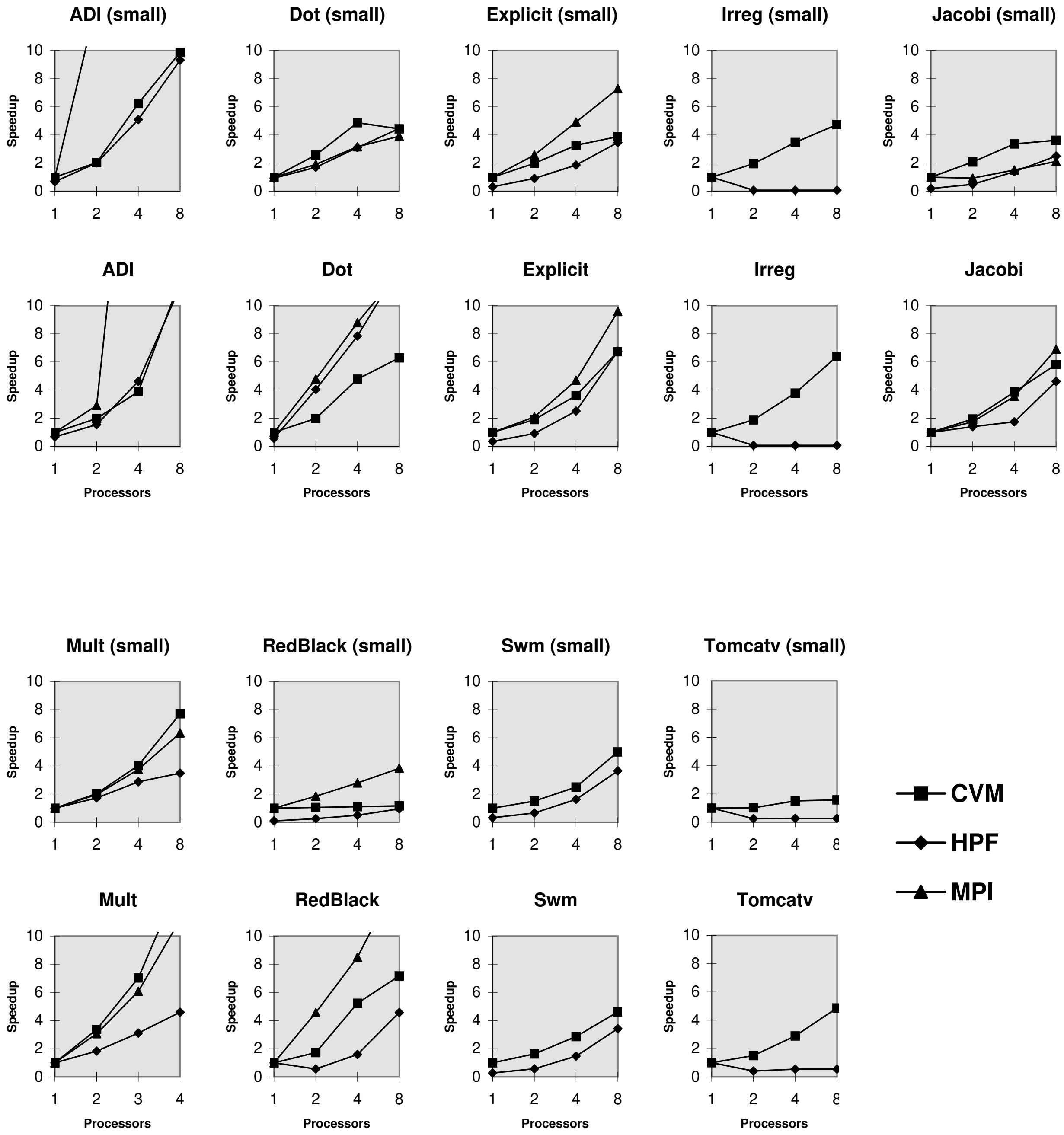

Figure 2 Speedups for DEC Alpha Cluster

4.4 Comparing CVM Performance against HPF and MPI

Figure 1 shows that on the IBM SP-2, HPF speedups were generally slightly higher than CVM and even MPI speedups. This indicates the IBM HPF compiler is quite powerful and is able to efficiently exploit low-level communication primitives. CVM speedups were nevertheless quite close to HPF speedups, with the major exceptions of SWM and TOMCATV which experience excessive paging during parallel execution. We plan on fixing this problem by tuning the page allocation policy in AIX.

Figure 2 shows that on the DEC Alpha cluster, CVM speedups match or exceed the HPF speedups in every case except DOT with the large data set. The DOT kernel has the highest incidence of reduction operations, which are somewhat more efficient on the HPF system. Otherwise, CVM almost always outperformed the corresponding HPF programs on the DEC cluster.

Examining the DEC Alpha results in more detail, we see that with the DEC HPF compiler programs execute slower than sequential Fortran 90 programs due to the overhead from invoking message passing routines. This overhead is significant, in many cases doubling the execution time of a one-processor HPF program. As a result overall speedups and execution times are reduced for HPF programs. However, because data is communicated efficiently (and only data actually used is transferred), we expect HPF programs to achieve good scalability in performance for larger numbers of processors. In comparison, CVM programs have virtually no overhead for one-processor execution.

One program that stands out is IRREG, since CVM was able to achieve speedups significantly better than HPF on both architectures. CVM was able to capture the pattern of irregular remote accesses at runtime and handle it almost as efficiently as for regular dense matrix accesses. In comparison, HPF compilers were unable to analyze the nonlocal accesses at compile time, resulting in inefficient execution. This example emphasizes the advantages of a combined compile/runtime approach for less regular computations.

Our experiments show that speedups of message passing programs using MPI were generally comparable to those of HPF and CVM programs on both parallel architectures. For the dense-matrix applications evaluated, it appears that both HPF and CVM are sufficiently efficient that it would require a fair amount of effort to customize message-passing programs for better performance under MPI.

Overall, the performance of CVM programs is comparable to that of the HPF programs for applications with sufficient computation. This result is encouraging, since most of the applications we examined have very regular access patterns, and hence represent the best case for HPF compilers. These results show that with enhanced software DSMs, the same performance can be achieved with much less compiler analysis for many applications on moderate-size parallel systems.

4.5 Detailed Evaluations

Figure 3 breaks CVM execution time down into five categories: application processing time, time spent waiting at barriers, miss handling time, time spent "flushing" data in our update protocol, and time spent in communication routines. Barrier wait 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 under CVM.

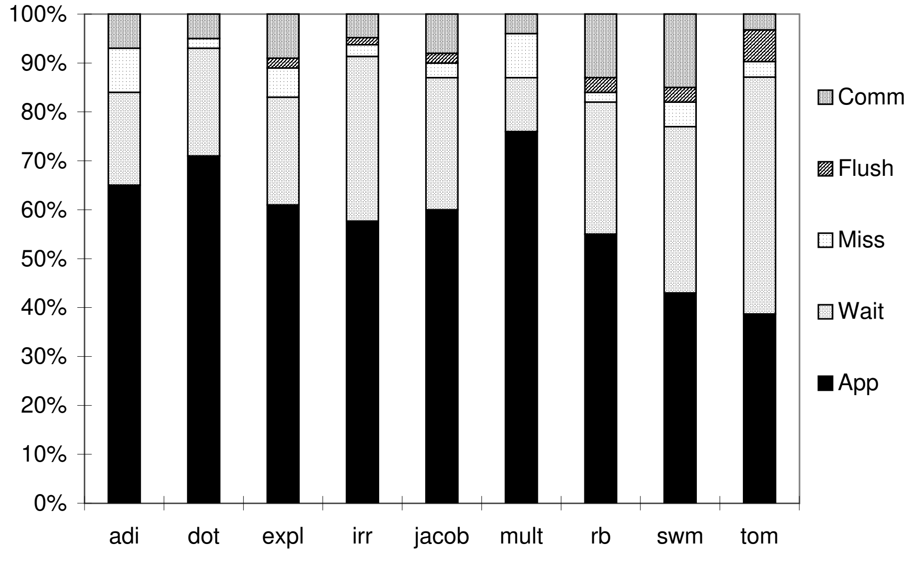

Figure 3 Breakdown of Execution Time

"Miss" time includes system time spent calling the fault handler and changing page protections, as well as all remote requests needed to validate shared pages. 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.

Recall that when enabled, our compiler automatically insert calls to DSM routines that mark address ranges to be kept coherent using a flush protocol; updates are *flushed* at barriers to eliminate nonlocal misses. For a given page, local modifications are flushed to processors named by the 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 created, pages invalidated, 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 for most of the programs, indicating that they have stable 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 "Useful Diffs" column shows that this is significant only for TOMCATV. The problem appears to be due to a less obvious disadvantage of barrier flushes, which 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. We intend to improve our compiler analysis to eliminate these unnecessary flushes. Despite the relatively large percentage of unused diffs, performance for TOMCATV is significantly improved because the large number of page misses eliminated.

Table 3 contains statistics describing executions with and without customized reduction support for the applications that use reductions. Without customized reductions, accumulations occur through mutually exclusive updates to shared memory, incurring lock synchronization overhead. The percentage of overall execution time spent waiting for lock access is listed in the "Sync Overhead" column. For applications with many reductions such as DOT, the time lost becomes a large fraction of total execution time. With customized reduction support, reduction records are created and piggybacked on barrier synchronization messages. No locks are needed to enforce mutual exclusion, eliminating synchronization overhead altogether. The results show that customizing reductions is quite effective for reducing access misses in those applications that have frequent reductions, such as DOT.

Table 4 shows the improvement in bandwidth requirements of the multiple-writer coherence protocol versus a single-writer protocol, as well as message and bandwidth totals for the applications with both optimizations turned on. Timings show that performance for some programs degrade significantly when using a single-writer coherence protocol. "Diff" requests are used to bring a page up to date. The message total reflects the fact that all messages except barrier flushes require a response. These numbers show CVM can handle communicating large amounts of data.

4.6 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. The programs in our study average a speedup of approximately 5 to 7 out of eight for large problem sizes. However, the super-linear speedup of two applications indicates that at least some of this speedup is due to caching effects. We plan to quantify this effect in the near future.

Our applications gain an average benefit of 14% from our compiler/DSM interface improvements for the large data sizes on eight processors. The optimizations have greater impact for more processors and smaller data sizes, and significantly improve performance for a few programs. Based on our experiences, we point out some likely avenues for further optimizations in the next section.

5 Additional Improvements

We believe that significant improvements are possible in both the compiler and the software DSM. We begin by discussing improvements to the compiler.

Better Update Classification
First, we anticipate doing a much better job on providing annotations for variables to guide flushing updates at barriers.

	Invals		Misses		Diffs		Msgs		Useful	Speedup
	w/o	w/	w/o	w/	w/o	w/	w/o	w/	Diffs	With
adi	655	270	655	0	649	649	1828	626	97%	19.6%
dot	105	105	105	0	99	100	6811	2968	100%	0.9%
expl	2538	241	2545	0	2215	2378	6616	2174	93%	7.8%
irr	1926	108	1933	107	559	560	8596	4098	99%	36.0%
jacobi	756	44	763	0	541	541	2548	1238	99%	6.8%
mult	135	2	135	0	129	130	508	334	100%	1.3%
rb	1008	64	1015	0	577	1081	4060	2462	99%	11.4%
swm	5071	114	16683	11669	5495	6148	56774	47176	99%	18.0%
tomcat	5834	74	5848	2	4024	7814	15454	5565	72%	30.8%

Table 2 Flush Protocol (IBM SP-2, 8 processors, large data sizes)

	Invals		Misses		Diffs		Msgs		Sync	Speedup
	w/o	w/	w/o	w/	w/o	w/	w/o	w/	Overhead	With
dot	950	105	950	105	944	99	6811	2968	23.7%	44.3%
irreg	2019	1926	2026	1933	922	559	10364	8596	6.3%	15.1%
jacobi	15336	15176	995	840	87	104	3738	2870	9.4%	12.5%
swm	5078	5071	16690	16683	5501	5495	56811	56774	0.0%	0.6%
tomcat	5850	5834	5864	5848	4040	4024	15882	15454	3.4%	5.4%

Table 3 Reduction Support (IBM SP-2, 8 processors, large data sizes)

Programs	Improvement vs. Single Writer	Messages				Bandwidth (kbytes)
		Barrier	Flush	Diff	Total	
adi	3610%	630	240	258	1128	532
dot	7%	2884	0	56	2940	297
expl	18%	1754	722	308	2784	7408
irreg	4%	1015	1851	214	3080	7572
jacobi	12%	1190	468	80	1738	4906
mult	0%	352	120	2332	2804	9747
rb	33%	2310	960	54	3324	5563
swm	1226%	1976	4132	24196	30304	100608
tomcat	444%	3542	1797	4	5343	20597

Table 4 Multiple Writer Communication Requirements (IBM SP-2, 8 processors)

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. Since our compiler does not yet detect phase changes, copysets would become poor predictors of future accesses for such applications. 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 essential for large, complex applications.

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 [1]. 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.

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 [13]. The compiler must also modify the code so data so nonlocal accesses are made to the buffers.

Message Library Support
Figure 3 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. Our port currently uses 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.

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.

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 [3, 7, 22], we are aware of only a few projects combining compilers and software DSMs. Bershad *et al.* maintain coherence by using a compiler to update a software dirty bit on shared-memory accesses [2]. Scales *et al.* designed Shasta, a software-only approach that supports fine-grain coherence through binary rewriting [24]. Using a number of optimizations, Shasta limits software overhead to within 5–35% for the Splash benchmarks on a DEC Alpha cluster. In comparison, CVM, like most software DSMs, relies on the virtual memory system to detect shared memory updates. Results, however, show that this is not a problem since 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 [21] on Typhoon, a Flexible-Shared-Memory machine implemented on top of a CM-5 [23]. 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 [5]. The authors point out that a compiler like SUIF can take advantage of the extensible coherence protocol to improve performance. Compared with their approach, we use a single general coherence protocol in the CVM for all applications, exploiting compile-time analysis to provide hints to the software DSM. The large number of customized coherence protocols they used for each application does not appear to be necessary for compiler-parallelized applications.

The SUIF compiler draws on a large body of work on techniques for identifying parallelism [12]. Previous researchers have examined shared-memory compilation issues such as improving locality [19] and reducing false sharing [26], but their techniques were mostly needed for single-writer hardware coherence protocols. Granston and Wishoff suggest a number of compiler optimizations for software DSMs [10]. 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 [20]. 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; we differ in requiring less analysis and by providing a more fine-grained API to control the behavior of individual pages.

Dwarkadas *et al.* recently described applying compiler analysis to explicitly parallel programs to improve the performance of software DSM [6]. By combining analysis in the ParaScope programming environment with TreadMarks, they were able to compute data access patterns at compile time and use it to help the runtime system aggregate communication and synchronization. Results for five programs were within 9% of equivalent HPF programs on the IBM SP-2. Compared to their system, the SUIF/CVM interface is targeted towards optimizing compiler-parallelized programs which are less tuned for software DSMs.

Finally, Chandra and Larus have preliminary results from a system combining the PGI HPF compiler and the Tempest software DSM system [4]. The PGI HPF compiler can generate either message-passing code or shared-memory code relying on Tempest. Preliminary results on a network of workstations connected by Myrinet indicates shared-memory versions of dense matrix programs achieve performance close to the message-passing codes generated. Tempest is significantly more efficient than message-passing for programs with irregular access patterns not analyzed at compile time. Unlike CVM, Tempest provides fine-grain access control for units smaller than a page [23]. However, since CVM supports multiple writers, the main performance advantage is in avoiding page faults traps for shared data. Large units of coherence can exploit spatial locality, so the PGI compiler can actually improve performance by using larger coherence units [4]. In comparison to PGI/Tempest, we implement and evaluate enhancements to the software DSM to improve performance. We are also able to demonstrate good performance on architectures with much longer communication latencies than the Myrinet interconnect, a more difficult task.

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 eliminate this restriction by combining a powerful shared-memory parallelizing compiler with an advanced software DSM system. Our results show that a few simple enhancements to the compiler/system interface can allow our system to approach the performance of commercially available HPF compilers and MPI message-passing programs. Our improvements: 1) combine synchronization and parallelism information communication on parallel task invocations, 2) employ customized routines for evaluating reduction operations, and 3) select a em flush protocol to pre-send data by flushing updates at barriers. Though these optimizations yield good speedups for program kernels with coarse-grain parallelism, performance for programs with smaller granularity of parallelism still has room for improvement. Nonetheless, our experiences lead us to believe that even a small amount of additional compiler analysis may allow our system to approach our long-term goal: effectively running applications that are too complex to be compiled directly to message-passing code.

References

[1] 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.

[2] 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.

[3] 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.

[4] S. Chandra and J.R. Larus. HPF on fine-grain distributed shared memory: Early experience. In *Proceedings of the Ninth Workshop on Languages and Compilers for Parallel Computing*, San Jose, CA, August 1996.

[5] 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.

[6] S. Dwarkadas, A. Cox, and W. Zwaenepoel. An integrated compile-time/run-time software distributed shared memory system. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, Boston, MA, October 1996.

[7] 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.

[8] Andrew Erlichson, Neal Nuckolls, Greg Chesson, and John Hennessy. SoftFLASH: Analyzing the performance of clustered distributed virtual shared memory. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[9] 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.

[10] 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.

[11] M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, D. Shields, K.-Y. Wang, W.-M. Ching, and T. Ngo. An HPF compiler for the IBM SP2. In *Proceedings of Supercomputing '95*, San Diego, CA, November 1995.

[12] 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.

[13] 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.

[14] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Preliminary experiences with the Fortran D compiler. In *Proceedings of Supercomputing '93*, Portland, OR, November 1993.

[15] P. Keleher. The relative importance of concurrent writers and weak consistency models. In *16th International Conference on Distributed Computing Systems*, Hong Kong, May 1996.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

[24] D. Scales, K. Gharachorloo, and C. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grained shared memory. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, Boston, MA, October 1996.

[25] Per Stenstrom. MP nodes don't need dedicated protocol processors. DRAFT: submitted for publication, October 1995.

[26] 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.