Unified Compilation Techniques for Shared and Distributed Address Space Machines

Chau-Wen Tseng, Jennifer M. Anderson, Saman P. Amarasinghe and Monica S. Lam

Computer Systems Laboratory
Stanford University
Stanford, CA 94305-4070

Abstract

Parallel machines with shared address spaces are easy to program because they provide hardware support that allows each processor to transparently access non-local data. However, obtaining scalable performance can be difficult due to memory access and synchronization overhead. In this paper, we use profiling and simulation studies to identify the sources of parallel overhead. We demonstrate that compilation techniques for distributed address space machines can be very effective when used in compilers for shared address space machines. Automatic data decomposition can co-locate data and computation to improve locality. Data reorganization transformations can reduce harmful cache effects. Communication analysis can eliminate barrier synchronization. We present a set of unified compilation techniques that exemplify this convergence in compilers for shared and distributed address space machines, and illustrate their effectiveness using two example applications.

1 Introduction

Until recently, large-scale parallel machines tended to only have distributed address spaces. Examples of such architectures include the Intel Paragon, Thinking Machines CM-5, and IBM SP-2. Programmers must painstakingly insert the proper message passing code to share data between processors. Recent advances in computer architecture have made it possible to build scalable machines that support a shared address space. Even though the memory on these machines is physically distributed across the processors, programs can simply issue memory load and store operations to access any data in the entire machine. Architectures like the Stanford DASH, KSR-1 and Convex Exemplar have coherent hardware caches that can automatically exploit the locality of reference in programs. Besides being easier to program, these machines have the advantage that they share the same programming model with small-scale multiprocessor workstations. Thus, software written for the shared address space model can run on the large-scale parallel machines as well as the low-end, but more pervasive workstations. The combination of ease of use and scalability of software is likely to make shared address space architectures the primary supercomputer architecture in the near future.

It is relatively easy to produce correct parallel programs for shared address space machines, since it is not necessary to explicitly manage communication for non-local data. However, experiences with these machines suggest that achieving scalable performance still requires a non-trivial amount of effort. The primary reason is that interprocessor communication, though hidden, is still quite expensive. Even though the remote memory access costs on these machines are significantly lower than message passing costs on distributed address space machines, enhancing a program's locality of reference can greatly improve program performance. As processor speeds are growing much faster than memory speeds, we expect the enhancement of data locality to remain significant in the future.

Traditionally, research in compilers for shared address space machines has focused mainly on the extraction of parallelism in sequential programs. Only recently have research compilers started to use some loop-level optimizations to enhance data locality and minimize communication [16, 26, 32]. On the other hand, the main focus in compilers for distributed address space machines is in managing the distributed data. From the distribution of the data, the compiler derives the computation assignment and the communication optimizations. This paper shows that parallelism detection is but the first step in parallelizing programs for shared address space machines. In fact, many analyses and transformations applied by compilers for distributed address space machines to generate *correct* programs can be profitably adapted by compilers for shared address space machines to generate *efficient* programs.

Throughout the paper, we attempt to substantiate our argument with experimental measurements. These experiments are based on the SUIF (Stanford University Intermediate Format) parallelizing compiler [31]. The SUIF compiler system has many of the common optimizations found in commercial compiler systems. A distinctive feature of this compiler system is that it is capable of finding parallelism across procedure boundaries. The compiler contains a large set of interprocedural analyses [12]: symbolic analysis, data dependence, array privatization [30], as well as reductions to both scalar and array variables. The compiler also contains some of the novel optimizations described in this paper. Since the SUIF compiler system is structured as a set of modular optimizations, we are able to disable different optimizations so as to assess the importance

This research was supported in part by ARPA contracts DABT63-91-K-0003 and DABT63-94-C-0054, an NSF Young Investigator Award, an NSF CISE Postdoctoral Fellowship in Experimental Science, and fellowships from Digital Equipment Corporation's Western Research Lab and Intel Corporation.

of individual techniques.

We start by showing the simulated performance of a set of programs on a hypothetical shared address space machine. Simulation is used so that we can measure and analyze the causes of the inefficiencies in these programs. The results indicate that simply finding parallelism in programs does not guarantee good performance. Analysis suggests that major causes of inefficiency include high synchronization costs and high average memory access times. This paper then presents a set of compiler techniques to address these problems. Many of the optimizations are commonly used in compilers for distributed address space machines. Finally, we perform two case studies to demonstrate the effectiveness of the compiler techniques.

2 Performance of Parallelizing Compilers

Experience with parallelizing compilers has shown that parallelism detection alone is not sufficient for achieving scalable performance for modern shared address space machines. To discover why speedups are lower than expected, we consider a recent simulation study by Torrie *et al.* that evaluates the impact of advanced memory systems on the behavior of compiler-parallelized programs [28]. The study analyzed the performance of a collection of programs from the SPEC, NAS and RICEPS benchmark suites that are successfully parallelized by the SUIF interprocedural parallelizer [12]. To quantify the success of parallelization, we define *parallel coverage* to be the percentage of instructions executed within parallel regions. Table 1 shows some of the programs included in the study; all these programs have a parallel coverage of at least 90% as measured by pixie on the Stanford DASH multiprocessor [20].

Program	Benchmark Suite	Description	% Parallel Coverage
APPBT	NAS	block-tridiagonal PDEs	100
APPSP	NAS	scalar-pentadiagonal PDEs	98
CGM	NAS	sparse conjugate gradient	91
ERLE64	MISC	ADI integration	98
HYDRO2D	SPEC	Navier-Stokes	98
MGRID	NAS	multigrid solver	95
ORA	SPEC	ray tracing	100
SIMPLE	RICEPS	Lagrangian hydrodynamics	95
SU2COR	SPEC	quantum physics	99
SWM256	SPEC	shallow water model	100

Table 1: Collection of Scientific Applications

For the programs in the study, the results of the interprocedural parallelizer are fed to a straightforward code generator. A master processor executes all the sequential portions of the code. The compiler parallelizes the outermost parallel loop in each loop nest, assigning to each processor a block of contiguous loop iterations of roughly equal size. Barriers are executed following each parallel loop to ensure that all processors have completed before the master continues.

The simulation study found that even if the compiler is able to detect significant parallelism, the performance obtained with this simple back-end strategy can be rather poor [28]. The target machine used for this study is an advanced directory-based cache-coherent non-uniform memory access (CC-NUMA) multiprocessor, similar to the FLASH multiprocessor [13]. It has a 200 MHz processor, a 100 MHz 256-bit local memory bus and a 200 MHz 16-bit wide mesh network interconnect. Each processor has

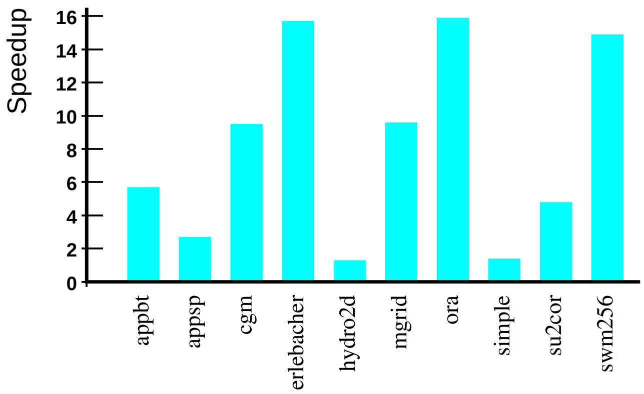


Figure 1: Application Speedups for 16 Processors

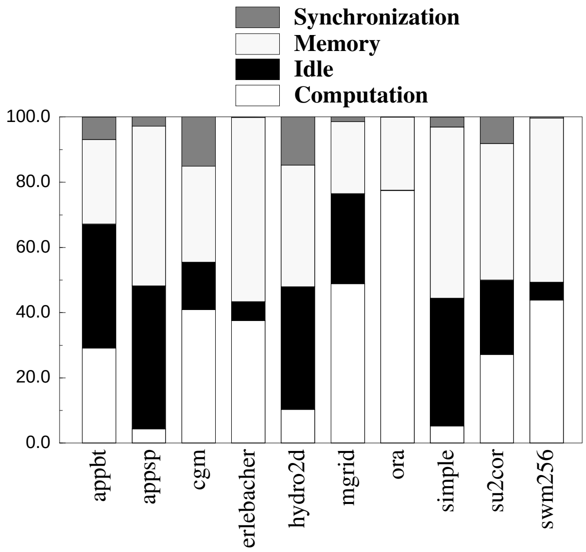


Figure 2: Categorization of Application Cycles, total over 16 processors

a single 128KB, 4-way set-associative cache, whose cache line size is 128 bytes.

Given certain assumptions about directory overhead for this machine model, the miss penalties work out to 88 cycles for a local miss, 324 cycles for a remote clean miss and 420 cycles for a dirty remote miss. The performance of our application programs on a 16-processor version of this simulated machine is shown in Figure 1. Despite the fact that all of the programs have uniformly high parallel coverage, the resulting speedups averaged only 8.2 and vary widely over each application, ranging from 1.3 (HYDRO2D) to 15.9 (ORA).

To analyze the causes of inefficiency, we break down the total simulated cycles in each application into four categories: useful computation cycles, idle time, memory access time and synchronization time. As shown in Figure 2, processor utilization is quite low in many cases. In fact, some of the applications (APPSP and

SIMPLE) spend under 10% of the time doing useful computation! On average, the processors spend about 33% on useful work, 23% on idle cycles, 39% cycles stalled on memory accesses and finally, 5% on synchronization and parallelization overhead.

3 Compilation Techniques

In this section, we carefully analyze each of the major factors of inefficiency and discuss useful compiler optimizations for dealing with them. We start by examining the choice of scheduling strategy and its impact on load balance, describe compiler techniques to minimize cache misses, and finally consider optimizations to reduce the remaining communication and synchronization costs.

3.1 Load Balance

Our first investigation focuses on the idle time in the parallel execution of these applications and examines the issue of load balance in these codes.

The experimental results discussed in Section 2 are based on a compiler that statically distributes the computation evenly across the processors. An alternative strategy is to dynamically assign the iterations to free processors as the execution proceeds. Dynamic scheduling has the advantage of better load balance, as the distribution of computation can adapt to the variations in the execution time of the iterations in the loop. Static scheduling, on the other hand, has the advantage of incurring a lower run-time overhead. Moreover, there tends to be more spatial locality in the computation assigned to each processor, as each processor gets to execute a contiguous block of iterations. While both static and dynamic scheduling have been used for shared address space machines, compilers for distributed address space machines use static scheduling exclusively.

We start by ignoring the memory subsystem and synchronization for the time being, and only concentrate on the distribution of instructions across the processors. In the following, we discuss three major factors for load imbalance. Figure 3 shows the ratio between the useful and idle cycles due to these factors. The instruction counts for the data in the figure were obtained using pixie. We ran the pixified programs on DASH with 16 processors and tallied the instruction counts for each processor. The data in this figure can only partially attribute for the idle cycles shown in Figure 2. Here we ran the entire program, whereas in the simulation study [28] a limited number of time steps were run over the full data set.

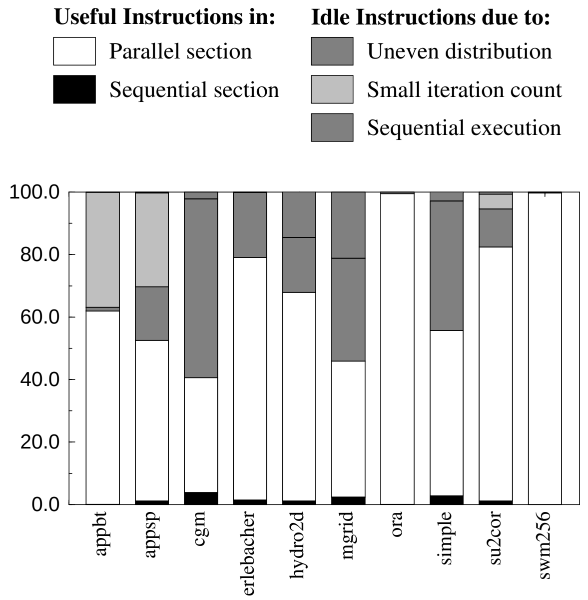


Figure 3: Sources of Load Imbalance

Sequential execution. As prescribed by Amdahl's law, parallel performance is heavily dependent on having a small sequential execution bottleneck. All processors but one are idle during the execution of those sections of code that have not been parallelized. Since there are 16 processors in our experiment, every unit of sequential execution time leads to a total of 15 units of idle cycles. This 15-fold amplification can create a significant number of idle cycles even if the parallel coverage of the applications is over 90%. The number of idle instructions due to the sequential execution bottleneck is significant for CGM, MGRID and SIMPLE, the programs with parallel coverage of 95% or less.

Small iteration count. Our investigation suggests that small iteration counts are another major source of load imbalance. Some processors are idle whenever there are fewer iterations in a parallel loop than there are processors, regardless of the scheduling scheme used. We used the pixie instruction counts to determine the number of iterations in each parallel loop, and to calculate the number of idle cycles induced in loops with less than 16 iterations. The results suggest that applications such as APPBT and APPSP suffer severely from having a small number of iterations for their parallel loops. Our current compiler parallelizes only the outermost parallel loop without regard to the number of iterations. It is possible to reduce the effect of small iteration counts by coalescing multiple parallel loops or by dynamically choosing an alternate loop to parallelize, if one exists, when the number of iterations is found to be small.

Uneven distribution. Finally, we focus on loops with at least 16 iterations, and study the distribution of the computation across the processors. Ideally, we would like to calculate the distribution of computation for each invocation of the parallel loop. However, pixie sums the total instructions within each parallel region over the entire program. This allows us to calculate only a lower bound on the idle instructions caused by load imbalance. We compute the idle instructions for each processor as follows. For each parallel region, we find the processor that executes the largest number of instructions. We then subtract the instructions executed by each processor from this maximum instruction count. The majority of the applications we looked at did not have much instruction imbalance. Only HYDRO2D and MGRID encountered noticeable numbers of idle instructions due to uneven computation distribution.

The results shown in Figure 3 satisfactorily explain the high idle-time components in Figure 2, with the exception of the programs HYDRO2D, SIMPLE and SU2COR. This is because so far we have ignored memory effects, and these programs have a non-trivial memory overhead component. Memory subsystem performance can also lead to load imbalance in parallel execution. Even though all processors may be assigned the same amount of computation, those with more cache misses will run longer and increase load imbalance. Dynamic scheduling can even out the imbalance by assigning more computation to processors that have higher cache misses in execution their computation. However, dynamic scheduling would also tend to exacerbate the memory subsystem performance problem, which may offset the advantage of a better load balance. A more direct solution is to improve the cache performance on these applications, which will also have the beneficial effect of minimizing the load imbalance. In general, static scheduling appears to work well for the applications found in the study.

3.2 Minimizing Cache Misses

There are two kinds of memory subsystem optimization techniques: minimizing cache misses, and hiding the cost of the remaining cache misses. We will study the former here, and address the latter together with synchronization optimization in the next section.

3.2.1 Causes of Cache Misses

Before discussing the compiler techniques useful for minimizing cache misses, we first analyze the causes of cache misses. *True sharing* cache misses occur whenever two processors access the same data word. Characteristics found in typical data caches can also introduce several other problems. First, data are transferred in fixed-size units known as *cache lines*; computations with spatial locality can take full advantage of the multiple-word transfer in each cache fill. However, if a processor re-uses a data item, the item may no longer be in the cache due to an intervening write access by another processor to a different word in the same cache line. This phenomenon is known as *false sharing*.

Another problematic characteristic of data caches is that they typically have a small set-associativity; that is, each memory location can only be cached in a small number of cache locations. Thus, even though the data accessed may be well within the *capacity* of the cache, *conflict misses* may still occur as a result of having different memory locations contend for the same cache location. Since each processor only operates on a subset of the data, the addresses accessed by each processor are often distributed throughout the shared address space, thus potentially leading to significant conflict misses.

In the simulation study of compiler-parallelized codes, the causes of cache misses were analyzed [28] and are shown in Figure 4. The experiment classifies each miss as either a *cold* miss, which occurs the first time each processor accesses data, a true sharing miss, a false sharing miss, and finally a *replacement* miss, which may either be a conflict miss or a capacity miss. The miss rate for each type of cache miss is measured as the percentage of all references; it is plotted along the Y-axis in Figure 4.

The results show that cache misses vary significantly across the programs. We observe that many programs have significant true sharing misses. In addition, the APPSP application has a significant amount of replacement misses, and SIMPLE has a significant amount of false sharing misses. Because of the high cache miss penalty, a program's performance is highly sensitive to the cache miss rate. For example, even though the cache miss rate in ORA is very small, less than 0.1%, the program still spends over 20% of its execution time in the memory subsystem. For programs with high miss rates, e.g. APPSP, HYDRO2D and SIMPLE, the machine spends much more

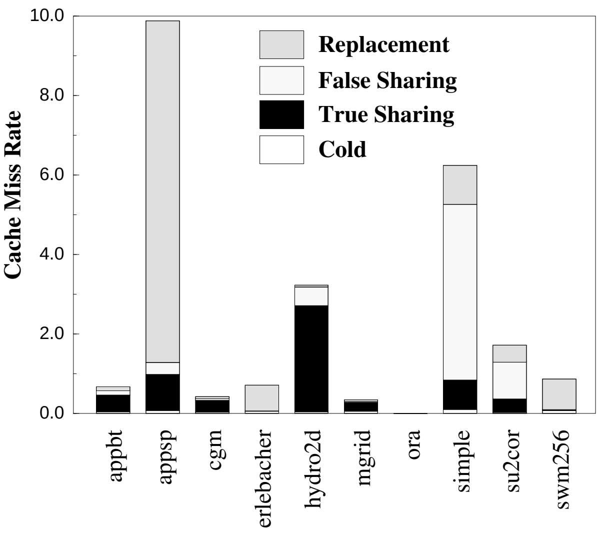


Figure 4: Causes of Cache Misses

time stalled for memory accesses than on useful computation, as shown in Figure 2.

The results suggest that compiler optimizations that maximize a program's locality and hence minimize true sharing are important. However, such optimizations alone are insufficient to use the memory system effectively. False sharing and replacement misses are also significant factors that must be addressed. As these misses are a function of how the data are laid out in the address space, it is possible to improve the cache performance by disregarding the data layout convention and customizing the layout for the specific program. We discuss optimizations to minimize true sharing, then data layout optimizations below.

3.2.2 Parallelism and Locality Optimizations

The goal of exploiting parallelism while minimizing essential communication (*i.e., true sharing*) is common to all parallel machines, be they distributed or shared address space machines. On distributed address space machines, the problem has been formulated as finding computation and data decomposition schemes across the processors such that interprocessor communication is minimized. This problem requires global analysis be applied across different loop nests to determine which loops to parallelize. A popular approach to this problem is to leave the primary responsibility to the user. The user specifies the data-to-processor mapping using a language such as HPF [14], and the compiler infers the computation mapping by using the owner-computes rule [15]. Recently, a number of algorithms for finding data and/or computation decompositions automatically have been proposed [3, 4, 11, 24].

For shared address space machines, there is no need to find the data decompositions per se, since it is not necessary to allocate arrays in separate local address spaces. However, to globally minimize true sharing across loop nests, the analysis must keep track of the data used by each processor. Thus, the algorithm used to minimize interprocessor communication for distributed address space machines can be directly applied to shared address space machines.

3.2.3 Data Layout Optimization

On a distributed address space machine, data decompositions are used directly to perform local memory allocations and local address calculations. They also provide valuable information to compilers for shared address space machines for minimizing false sharing and cache conflicts. We observe that placing the data accessed by each processor contiguously in the shared address space tends to enhance spatial locality, minimize false sharing and also minimize conflict misses. Once the data decomposition is known, a simple algorithm exists to transform the data layout to make the region accessed by each processor contiguous [2].

Sometimes it is necessary to change the data decompositions dynamically as the program executes to minimize communication. A compiler for shared address space machines may choose to change the data layout accordingly at run time. Alternatively, it can simply choose to implement the dominant data layout and rely on the hardware to move the data when necessary.

3.3 Reducing Synchronization and Communication Overhead

After globally minimizing both true and false sharing through memory optimizations, our next focus is the efficient implementation of the remaining synchronization and communication operations. Before discussing the details, we first overview the control structures of parallelized code for shared and distributed address space machines.

Parallelizing compilers for shared address space machines typically employ a *fork-join* model, where a single master thread executes the sequential portions of the program, assigning computation to worker threads when parallel loops are encountered. Typically, synchronization between processes is achieved by having processors participate in a barrier at the end of each parallel loop. Occasionally, point-wise synchronization operations are used to implement do-across loops. Interprocessor communication of data occurs implicitly whenever processors issue load and store operations to remote data.

In contrast, distributed-memory compilers generate code according to a *single-program, multiple-data* (SPMD) model, where all threads execute the entire program. Sequential computation is either replicated or explicitly guarded to limit execution to a single thread. When a parallel loop is encountered, each thread executes a portion of the computation based on their processor IDs. On distributed address space machines, synchronization is tied to communication as a processor automatically stalls on receive operations. Thus, unlike barriers, only interacting processors need to synchronize with each other. Furthermore, significant attention is paid to amortize the cost of issuing a message and to overlap the latency of communication with useful computation.

As the cost of synchronization on shared address space machines becomes nontrivial relative to the processor speed, many of the optimizations used only in distributed address space machines also become applicable.

3.3.1 Minimizing Synchronization

Barriers required after each parallel loop in current shared address space programs can impose significant overhead for two reasons. First, executing a barrier has some run-time overhead that typically grows quickly as the number of processors increases. Second, executing a barrier requires all processors to idle while waiting for the slowest processor; this effect results in poor processor utilization when processor execution times vary. Eliminating the barrier allows perturbations in task execution time to even out, taking advantage of the loosely coupled nature of multiprocessors. Barrier synchronization overhead is particularly significant when attempting to use many processors, since the interval between barriers decreases as computation is partitioned across more processors. Figure 2 shows that some of these programs incur significant overhead in synchronization operations.

If the data and computation decompositions successfully eliminate communication between loop nests, then it is not necessary for processors to synchronize at the end of every parallel loop. It is thus desirable to increase the autonomy of the computation on each processor in a fork-join model using ideas found in the SPMD programming model on distributed address space machines. Sequential parts of the program are executed by a single master thread, as in traditional compilers for shared address space machines. However, by employing a hybrid SPMD model, the computation assigned to each worker processor can span a series of parallel loops [7]. Barriers may be eliminated if they are unnecessary, or they may be replaced with more efficient point-to-point synchronization operations [29].

Barrier optimization requires analysis of dependences across loop boundaries. While inter-loop analysis is not used in conventional compilers for shared address space machines, such analysis is required to generate the necessary communication code for distributed address space machines. Simply, synchronization between a pair of processors is necessary whenever there is need for interprocessor communication on distributed address space machines. One important difference, however, is that a compiler for shared address space machines can simply insert a barrier between any pair of loops that cannot be analyzed statically. A compiler for distributed address space machines, on the other hand, must insert the suitable code to ensure that all the data needed are transferred.

3.3.2 Communication Optimizations

Because of the high communication cost in distributed address space machines, a lot of research effort has been devoted to the optimization of communication. Given a compile-time data and computation decomposition, the compiler applies *communication analysis* to track the movement of data between processors. This information can then be used to perform a number of communication optimizations. Many of these optimizations are also applicable to shared address space machines, even though they tend to have lower communication costs.

Latency hiding. Compilers for distributed address space machines try to overlap communication and computation to hide high communication latencies. Due to the high cache miss penalty, even recent microprocessors include *prefetch* operations in their instruction sets so that cache miss penalties can be overlapped with computation of other data. Because prefetches incur additional software instruction overhead, it is thus necessary to avoid issuing unnecessary prefetch operations [23]. Communication analysis similar to what is used for distributed address space machines is thus also useful for compilers for shared address space machines. Prefetch algorithms are different in some ways. For example, we must also issue prefetches to local data that miss in the cache. Also, prefetches are nonbinding, thus a processor can optimistically prefetch data without knowing for sure that the data are ready. The hardware would automatically keep the data coherent if the values change subsequently.

Amortization of communication overhead. An important optimization used in compilers for distributed address space machines is to aggregate communication into large messages so as to amortize the high message passing overhead. This analysis may be applied for shared address space machines that provide efficient block data transfer [13, 17]. This technique is also useful for moving a large collection of data on separate caches lines. The compiler can pack the data to be communicated into contiguous locations, and thus reduce the number of cache lines transferred. This technique may be important for machines with very long cache lines.

Eliminating round-trip communication. In a typical message passing program, the sender initiates send operations without prompting from the receiving processor. On the other hand, shared address space programs generally require the consumer to initiate the data transfer. Thus every data transfer requires round-trip communication between the sending/receiving processors. For computations where the compiler can calculate the identity of the processors at compile-time, the compiler can reduce latency to a one-way communication, by pushing data instead of pulling data, just as in message passing. For instance, DASH provides an operation where the producer can initiate updates to remote copies to reduce latency. However, this application of distributed address space compilation technique has been shown to be useful mainly for implementations of global synchronization primitives such as barriers and accumulations [17].

4 Case Studies

We have implemented many of the optimizations described in this paper in the SUIF compilation system [31]. The SUIF compiler takes as input either Fortran or C, performs multiple optimization passes, then generates C code that can be linked with a portable run-time library. Currently only the shared address space machine code generator is in place. Figure 5 illustrates the basic architecture of the compiler.

To evaluate the impact of our optimizations on shared address space machines, we present case studies for LU decomposition and the SPEC benchmark TOMCATV. All of the optimizations used in the case studies have been implemented and are applied automatically by the SUIF compiler. We present simulation results to demonstrate how each optimization affects program behavior. We also examine speedups on the Stanford DASH multiprocessor to evaluate how the optimizations affect performance on a real machine.

4.1 Experimental Setup

We obtained statistics on the memory and synchronization behavior of each example program with the MemSpy simulator used by Torrie *et al.* [28]. We simulated two cache configurations for 16 processors: 128KB, 128 byte cache line, and either direct-mapped or 4-way set associative LRU caches. The 4-way version was the same cache model used for collecting our previous simulation results.

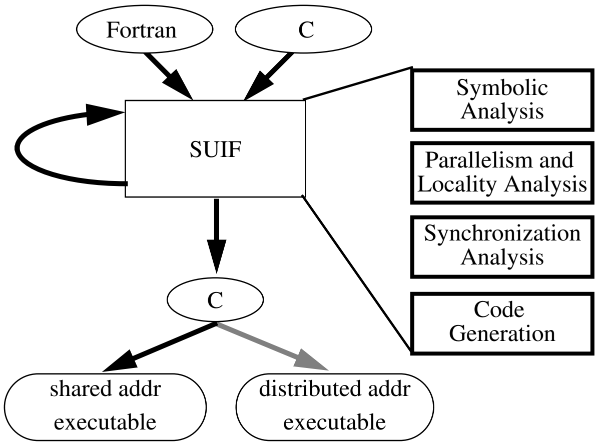


Figure 5: SUIF Compiler Architecture

Speedups for each program were obtained on the DASH multiprocessor, a cache-coherent NUMA architecture [20]. The machine we used for our experiments consists of 32 processors, organized into 8 clusters of 4 processors each. Each processor is a 33MHz MIPS R3000, and has a 64KB first-level cache and a 256KB second-level cache. Both the first and second-level caches are direct-mapped and have 16 byte lines. Each cluster has 28MB of main memory. A directory-based protocol is used to maintain cache coherence across clusters. We compiled the C programs produced by SUIF using `gcc` version 2.5.8 at optimization level `-O3`. Performance is presented as speedups over the best sequential version of each program.

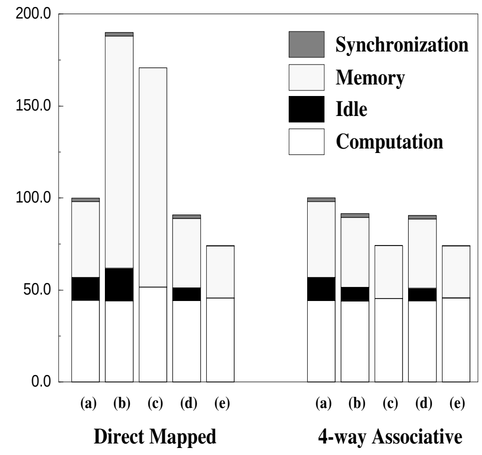
4.2 Evaluation

The basic SUIF compiler has capabilities similar to traditional shared-memory compilers such as KAP [18]. These abilities include parallelizing outer loops, scheduling iterations across processors evenly in contiguous blocks, and performing program transformations such loop permutation to enhance data locality. We consider this system to be our baseline compiler, and label its performance as BASE in each figure.

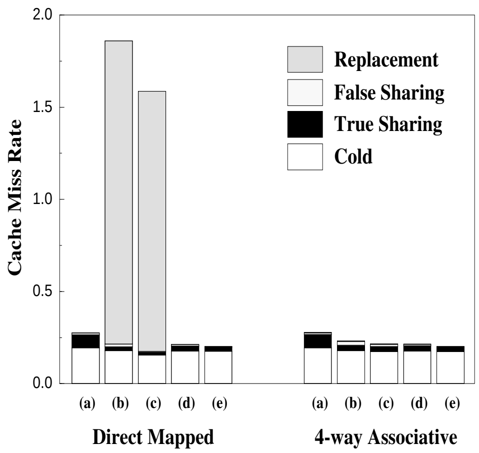
By selectively executing SUIF optimization passes, we generated multiple versions of each program. The versions labeled with COMP DECOMP are produced by running the computation decomposition phase of the compiler to co-locate data and computation from section 3.2.2. The versions labeled SYNCH show programs where the synchronization optimizations from section 3.3.1 have been applied. The versions labeled DATA TRANSFORM are generated by running the data transformation optimizations described in section 3.2.3. To evaluate the importance of different optimizations, we compared the different versions of each program against each other and against the code produced by the base compiler.

4.2.1 LU Decomposition

We first examine LU decomposition without pivoting. Program behavior for the 256×256 version of LU decomposition is shown in Figure 6, speedups for both versions are presented in Figure 7. For LU decomposition, the baseline compiler schedules the iterations of



(i) Cycle Distribution



(ii) Cache Statistics

(a) base
(b) comp decomp
(c) comp decomp + synch
(d) comp decomp + data transform
(e) comp decomp + synch + data transform

Figure 6: LU Decomposition Statistics, for a 256x256 Array

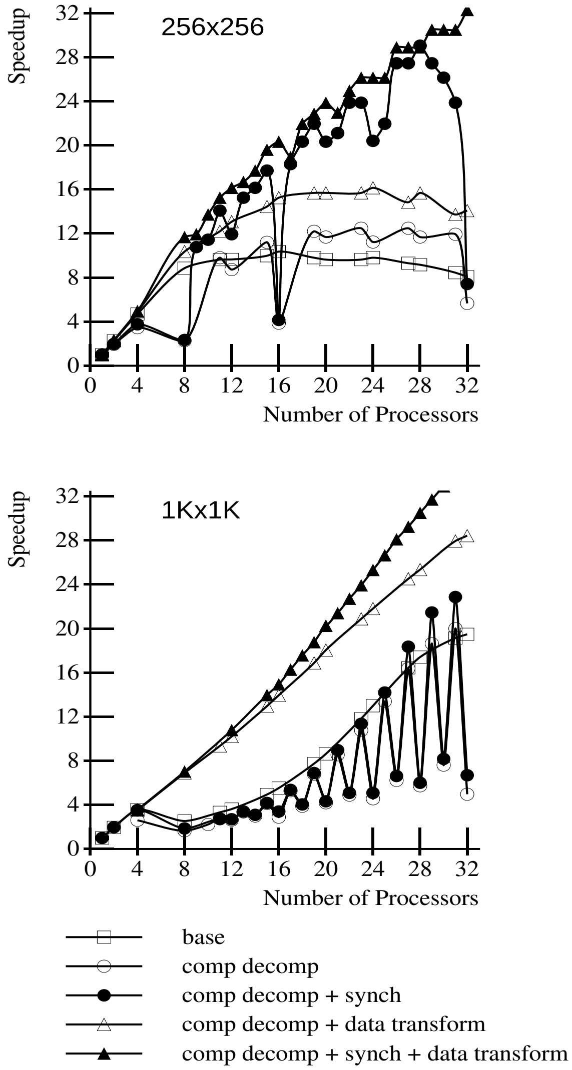


Figure 7: LU Decomposition Speedups

```
broadcast
sender = 0
do k = 1,n
  if (k ≠ 1) and (myid ≠ sender) then
    wait_counter
  sender = mod(k, nprocs)
  count = count + 1
  do j = lb, ub
    a(j,k) = a(j,k) / a(k,k)
    do i = k+1,n
      a(i,j) = a(i,j) - a(k,j) * a(i,k)
    enddo
    if (j == k+1) and (myid == sender) then
      increment_counter
  enddo
enddo
barrier
```

Figure 8: LU Decomposition Example Output

the outermost parallel loop uniformly across processors in a block-wise fashion. A barrier is placed after the parallel loop and used to synchronize between iterations of the outer sequential loop. Since the bounds of the parallel loop are a function of the outer sequential loop, they act as a sliding window, causing each processor to access different data each time through the outer loop.

When the computation decomposition optimization is invoked, it assigns all operations on the same row of data to the same processor. To achieve high processor utilization the rows are distributed in a cyclic (round-robin) manner. This distribution yields good load balance and has good data reuse. However, local data in each processor are scattered in the global address space, causing self-interference between the rows of the array. A large number of conflict misses results.

With synchronization optimization, the compiler is also able to replace the barrier following the parallel loop with a counter incremented on each iteration of the outer sequential loop by the producing processor. Processors requiring the data wait for the counter to be incremented before proceeding. Figure 8 illustrates how the barrier has been replaced by a counter. No data transformation is needed for the base version of LU decomposition, since each processor already accesses contiguous blocks of the array. However, for the optimized programs data transformation is needed to map each processor's sections of the array into a contiguous region to avoid self-interference. Without this optimization, the performance is very sensitive to the number of processors. After transforming the data, the performance stabilizes and we achieve consistently good performance.

The first graph in Figure 6 displays the computation, idle, memory, and synchronization cycles spent by each program, scaled by the cycles in the base version of the program. The second graph in the figure presents the cache miss rate for cold, true sharing, false sharing, and replacement misses as a percentage of all references. Results are provided for both direct-mapped and 4-way set associative versions of the baseline 128 KB, 128 byte line cache.

Simulations show that the cache miss rate for the base version of LU decomposition is low, due to the relatively small problem size. The most significant source of cache misses other than cold misses is true sharing misses. Applying computation decomposition optimizations reduces the rate of true sharing misses, but for the direct-mapped cache dramatically increases the number of replacement misses due to cache conflicts. The increase in turn leads to higher memory cycles. The 4-way associative cache does not encounter the severe problem with conflict misses.

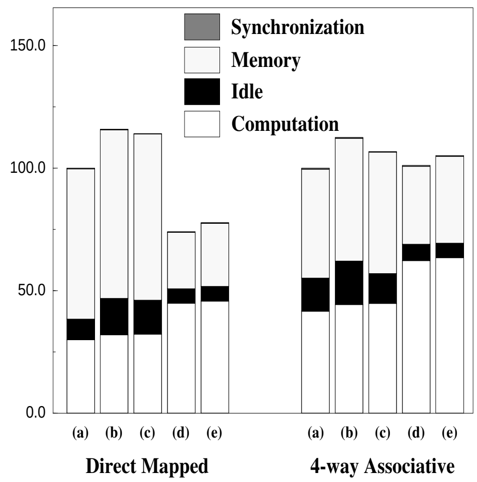
For the direct-mapped cache, data transformations to make local data contiguous is successful in eliminating these replacement misses. The optimized program yields memory overhead lower than the base version of the compiler. Synchronization optimization is also quite effective, eliminating synchronization overhead and reducing the number of idle cycles.

To see how these optimizations affect program performance, consider the speedups for each version of LU decomposition displayed in Figure 7. We see that both synchronization optimizations and data transformations are necessary for achieving good performance on LU decomposition. It is particularly interesting to note the variations in performance of LU decomposition when cache conflicts are exacerbated by running programs on an even number of processors. Superlinear speedup is observed for the fully optimized LU on DASH with larger problem sizes. This is due to the fact that once the data is partitioned among enough processors, each processor's working set fits into local memory.

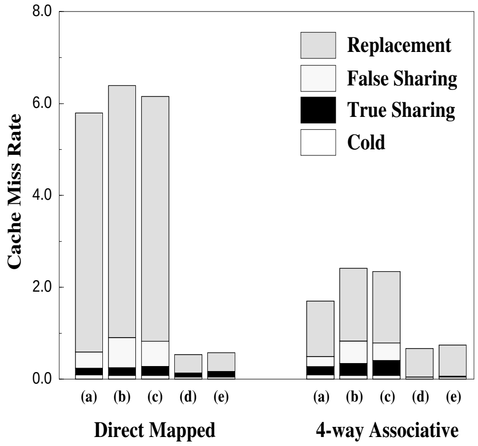
4.2.2 Tomcatv

We now look at our next case study, TOMCATV, a Fortran mesh generation program from the SPEC92 floating-point benchmark suite. Figure 9 displays memory and synchronization statistics for TOMCATV and Figure 10 shows the speedups.

TOMCATV contains several loop nests that have dependences across the rows of the arrays and other loop nests that have no dependences. The base version always parallelizes the outermost parallel loop. In the loop nests with no dependences, each processor thus accesses a block of array columns. In the loop nests with row dependences, each processor accesses a block of array rows. As a result of this inconsistency, there is little opportunity for data reuse across loop nests. Also, there is poor cache performance in the row-dependent loop nests because the data accessed by each processor is not contiguous in the shared address space.



(i) Cycle Distribution



(ii) Cache Statistics

(a) base
(b) comp decomp
(c) comp decomp + synch
(d) comp decomp + data transform
(e) comp decomp + synch + data transform

Figure 9: Tomcatv Statistics

In comparison, computation decomposition analysis selects a distribution so that each processor will always access a block of array rows. The row-dependent loop nests still execute completely in parallel, but potentially require executing inner loops in parallel. The COMP DECOMP version of TOMCATV has fewer true sharing misses than base version. However, the speedups are still poor, due to the large number of false sharing and replacement misses caused by cache conflicts.

Figure 9 shows that moving from a direct-mapped to a 4-way set associative cache reduces conflict misses for both the base and

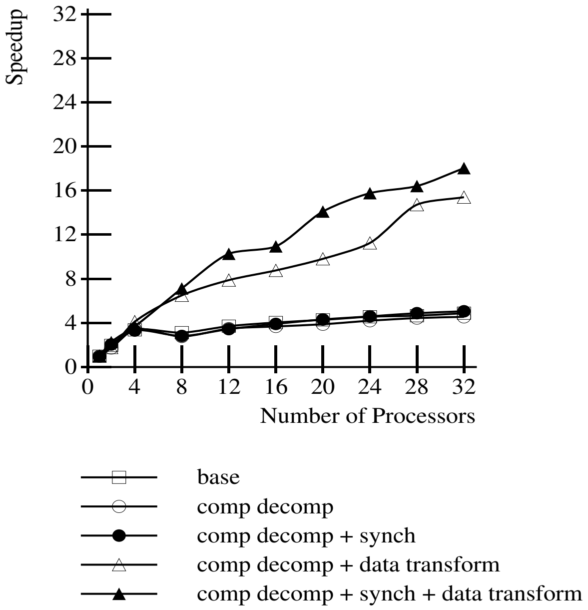


Figure 10: Tomcatv Speedups

COMP DECOMP versions of the program, but a significant amount still remains. To improve spatial locality, eliminate false sharing, and reduce cache conflicts, the data transformation optimization reorganizes the arrays in TOMCATV so that the rows accessed by each processor are contiguous in memory. This optimization results in good cache performance by eliminating nearly all the false-sharing and most of the replacement misses caused by cache conflicts.

The baseline SUIF compiler can perform a form of synchronization optimization for perfectly nested loops; it moves barriers out of sequential loops when data dependences show the inner parallel loop may be permuted to an outer position without modifying its loop bounds. This condition is similar to the *strip-mine and interchange* transformation used to increase granularity of parallelism while preserving data locality [16, 32].

For TOMCATV, this optimization can be applied to the COMP DECOMP versions of the program to move barriers following inner parallel loops outwards. More advanced synchronization optimizations are able to eliminate several barriers between adjacent parallel loop nests, reducing the number of barrier executions by half. Once all three optimizations are put together, the program performs well, achieving high speedups.

4.3 Summary

These case studies show that for certain applications on shared address space machines, distributed address space compilation techniques are essential for achieving high performance for larger numbers of processors. Of the optimizations we evaluated through case studies, we find that co-locating data and computation is the most useful for achieving high performance. Reducing false sharing is very significant in certain cases. Synchronization optimizations grow to be important as the number of processors increases.

5 Related Work

Our work builds upon shared-memory compiler algorithms for identifying parallelism [5, 12, 30] and performing program transformations [32, 33], as well as distributed-memory compilation techniques to select data decompositions [3] and explicitly manage address translation and data movement [1, 15].

Many previous researchers have examined performance issues on shared-memory architectures. Singh *et al.* [25] applied optimizations similar to those we considered by hand to representative computations in order to gain good performance. Others evaluated the benefits of co-locating data and computation [9, 22], as well as false sharing [6, 8, 10, 21, 27]. These approaches focused on individual optimizations and were generally applied by hand. In contrast, we have shown how they may be implemented in a compiler by adapting well-known techniques from distributed-memory compilers for shared-memory machines.

Larus has compared implementing global address spaces in software using a distributed-memory compiler compared to hardware-based implementations [19]. He speculates that distributed-memory compilers are desirable because they can more closely exploit underlying architectural features in certain key cases; however, shared-memory hardware is desirable in the cases where the compiler fails. Compared to his work we examine actual instances of compiler techniques and evaluate their impact on performance.

Cytron et al. [7] were the first to consider mixing fork-join and SPMD models. Their goal was to eliminate thread startup overhead and increase opportunities for privatization; they carefully considered safety conditions. In comparison, we use local SPMD regions to enable compile-time computation partition and synchronization elimination through communication analysis. Researchers using the Alewife multiprocessor compared the benefits of message-passing and shared memory [17]; they found message passing to be advantageous mainly for improving synchronization primitives by coupling synchronization and data movement.

6 Conclusions

In this paper, we examined issues in compiling for shared address space architectures. Despite their programmability, it can be difficult to achieve scalable speedups. Simulation studies identify memory and synchronization costs as major barriers to performance. Because they share many concerns for achieving high performance, we find many compilation techniques required to ensure correct execution on distributed address space machines can be adapted for achieving efficient performance on shared address space machines. Many of these optimizations described in this paper have been implemented in the SUIF compilation system. Experiments on the DASH multiprocessor show impressive improvements for some case studies. Our results seem to indicate a convergence of compilers for shared and distributed address space machines.

Acknowledgements

The authors wish to thank members of the Stanford SUIF compiler group for building and maintaining the software infrastructure used to implement this work. We acknowledge the Stanford DASH group for providing the hardware used in our experiments. Finally, we are very grateful to Evan Torrie, Margaret Martonosi, and Mary Hall

for the detailed results from their simulation study as well as the simulation technology used for our case studies.

References

[1] S. P. Amarasinghe and M. S. Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 126–138, Albuquerque, NM, June 1993.

[2] J. M. Anderson, S. P. Amarasinghe, and M. S. Lam. Data and computation transformations for multiprocessors. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, July 1995.

[3] J. M. Anderson and M. S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 112–125, Albuquerque, NM, June 1993.

[4] B. Bixby, K. Kennedy, and U. Kremer. Automatic data layout using 0-1 integer programming. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 111–122, Montreal, Canada, August 1994.

[5] 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.

[6] W. J. Bolosky and M. L. 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)*, pages 57–71, San Diego, CA, September 1993.

[7] R. Cytron, J. Lipkis, and E. Schonberg. A compiler-assisted approach to SPMD execution. In *Proceedings of Supercomputing '90*, pages 398–406, New York, NY, November 1990.

[8] S. J. Eggers and T. E. Jeremiassen. Eliminating false sharing. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages 377–381, St. Charles, IL, August 1991.

[9] J. Fang and M. Lu. A solution of cache ping-pong problem in RISC based parallel processing systems. In *Proceedings of the 1991 International Conference on Parallel Processing*, St. Charles, IL, August 1991.

[10] E. D. 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 and P. Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193, March 1992.

[12] 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*, pages 650–655, San Francisco, CA, February 1995.

[13] 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)*, pages 274–284, October 1994.

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

[15] 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.

[16] K. Kennedy and K. S. McKinley. Optimizing for parallelism and data locality. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

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

[18] Kuck & Associates, Inc. *KAP User's Guide*. Champaign, IL 61820, 1988.

[19] J. Larus. Compiling for shared-memory and message-passing computers. *ACM Letters on Programming Languages and Systems*, 2(1–4):165–180, March–December 1993.

[20] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH prototype: Implementation and performance. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 92–105, Gold Coast, Australia, May 1992.

[21] H. Li and K. C. Sevcik. NUMACROS: Data parallel programming on NUMA multiprocessors. In *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, pages 247–263, San Diego, CA, September 1993.

[22] 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.

[23] T. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 62–73, Boston, MA, October 1992.

[24] T. J. Sheffler, R. Schreiber, J. R. Gilbert, and S. Chatterjee. Aligning parallel arrays to reduce communication. In *Frontiers '95: The 5th Symposium on the Frontiers of Massively Parallel Computation*, pages 324–331, McLean, VA, February 1995.

[25] J.P. Singh, T. Joe, A. Gupta, and J. L. Hennessy. An empirical comparison of the Kendall Square Research KSR-1 and Stanford DASH multiprocessors. In *Proceedings of Supercomputing '93*, pages 214–225, Portland, OR, November 1993.

[26] O. Temam, E. D. Granston, and W. Jalby. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Proceedings of Supercomputing '93*, pages 410–419, Portland, OR, November 1993.

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

[28] E. Torrie, C-W. Tseng, M. Martonosi, and M. W. Hall. Evaluating the impact of advanced memory systems on compiler-parallelized codes. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, June 1995.

[29] C-W. Tseng. Compiler optimizations for eliminating barrier synchronization. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, July 1995.

[30] 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.

[31] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 1994.

[32] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.

[33] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. The MIT Press, Cambridge, MA, 1989.