Compiler Optimizations for High Performance Architectures

Hwansoo Han, Gabriel Rivera, Chau-Wen Tseng

Department of Computer Science
University of Maryland
College Park, MD 20742

Abstract

We describe two ongoing compiler projects for high performance architectures at the University of Maryland being developed using the Stanford SUIF compiler infrastructure. First, we are investigating the impact of compilation techniques for eliminating synchronization overhead in compiler-parallelized programs running on software distributed-shared-memory (DSM) systems. Second, we are evaluating data layout transformations to improve cache performance on uniprocessors by eliminating conflict misses through inter- and intra-variable padding. Our optimizations have been implemented in SUIF and tested on a number of programs. Preliminary results are encouraging.

1 Introduction

Our goal at the University of Maryland is to support efficient machine-independent programming of advanced architectures. Users desire the ability to write programs that can run well on a variety of computers, since such programs are portable and protect software investment. However, modern processor architectures are quite varied and complex. To achieve high performance, it is critical that the compiler generates code that can efficiently utilize the underlying hardware. To bridge the gap between applications, operating systems, and advanced architectures, the compiler must be able to perform advanced analyses and transformations. We are currently pursuing research in two areas: compiling for software distributed-shared-memory (DSM) systems and data layout optimizations for high performance architectures.

2 Reducing Synchronization Overhead

We begin by looking at compiler optimizations for reducing synchronization overhead, These optimizations are examined in the context of a prototype system [17] extending SUIF [11] to target the CVM [15] software distributed-shared-memory (DSM). Recent studies indicate such systems can approach the performance of current message-passing HPF compilers and explicit message-passing MPI programs [6, 17]. However, load imbalance and synchronization overhead were identified as key sources of inefficiency.

Execution times for five SUIF-parallelized applications on a 16 processor IBM SP-2 are shown in Figure 1. Total execution time is split into application processing time, OS overhead, communication cost, sequential wait (idle time waiting for next parallel task), load imbalance (idle time waiting for completion of current parallel task), and barrier overhead (time spent executing barrier code). We found while not much time is spent explicitly executing barriers (1.5% of execution time), idle processor time in the form of sequential wait and load imbalance comprises a large percentage of total execution time (14% and 27%, respectively).

This research was supported by NSF CAREER Development Award ASC9625531 and NSF Award CCR9711514. The IBM SP-2 and DEC Alpha Cluster were provided by NSF CISE Institutional Infrastructure Award CDA9401151 and grants from IBM and DEC.

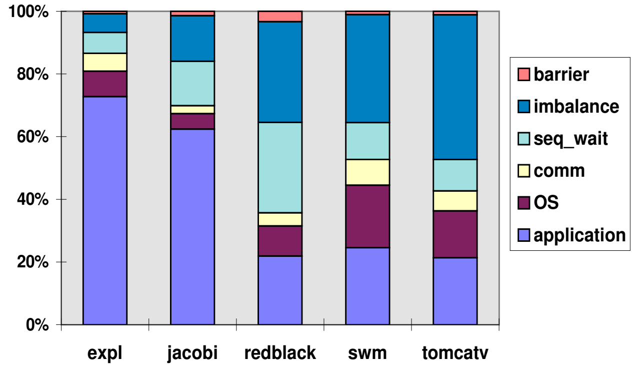

Figure 1: Breakdown of Execution Time (16 Processor SP-2)

The compiler-generated code is balanced computationally, but the underlying DSM and OS effectively add random (and unequal) delays. The extra overhead caused by processing coherence traffic at synchronization points also contributes to overhead. It is clear from these measurements that reducing sequential wait and load imbalance caused by synchronization overhead is important for achieving good performance.

2.1 Background

2.1.1 CVM Software DSM

CVM is a software DSM that supports coherent shared memory for multiple protocols and consistency models [15]. It is written entirely as a user-level library and runs on most UNIX-like systems. Its primary coherence protocol implements a multiple-writer version of *lazy release consistency* [16], a derivation of *release consistency*. Release consistency allows a processor to delay making modifications to shared data visible to other processors until special *acquire* or *release* synchronization accesses occur. Lazy release consistency postpones the propagation of modifications further; updates to shared data do not have to be made visible to a processor until the next time that processor acquires a released synchronization variable. Consistency information in CVM is piggybacked on synchronization messages. Multiple updates are also aggregated in a single message where possible.

2.1.2 SUIF/CVM Interface

SUIF was retargeted to generate code for CVM by providing a run-time interface based on CVM thread creation and synchronization primitives. Performance was improved by adding customized support for reductions, as well as a *flush update* protocol that at barriers automatically sends updates to processors possessing copies of recently modified shared data [17]. Analysis needed for the flush update protocol is simpler than communication analysis needed in HPF compilers; the compiler locates data likely to be communicated in a stable pattern, then insert calls to DSM routines

{ Default Model }

```
DOALL I = 1,N
  ...
ENDDO
DOALL J = 1,N
  ...
ENDDO
```

⇓

```
broadcast
DO I = LB1,UB1
  ...
ENDDO
barrier
broadcast
DO J = LB2,UB2
  ...
ENDDO
barrier
```

(A)

{ Dependence Analysis }

```
DOALL I = 1,N
  A(I) =
ENDDO
DOALL J = N+1,2*N
  B(J) = A(J)
ENDDO
```

⇓

```
broadcast
DO I = LB1,UB1
  A(I) =
ENDDO
DO J = LB2,UB2
  B(J) = A(J)
ENDDO
barrier
```

(B)

{ Communication Analysis }

```
DISTRIBUTE (BLOCK) :: A,B
DOALL I = 1,N
  A(I) =
ENDDO
DOALL J = 1,N
  B(J) = A(J)
ENDDO
```

⇓

```
broadcast
DO I = LB1,UB1
  A(I) =
ENDDO
DO J = LB1,UB1
  B(J) = A(J)
ENDDO
barrier
```

(C)

{ Nearest-Neighbor }

```
DISTRIBUTE (BLOCK) :: A,B
DOALL I = 1,N
  A(I) =
ENDDO
DOALL J = 1,N
  B(J) = A(J-1) + A(J+1)
ENDDO
```

⇓

```
broadcast
DO I = LB1,UB1
  A(I) =
ENDDO
sync_with_neighbors
DO J = LB1,UB1
  B(J) = A(J-1) + A(J+1)
ENDDO
barrier
```

(D)

Figure 2: Optimization Examples

to apply the flush protocol for those pages at the appropriate time.

2.1.3 Compile-time Barrier Elimination

The fork-join model used by shared-memory compilers is flexible and can easily handle sequential portions of the computation; however, it imposes two synchronization events per parallel loop as shown in Figure 2(A). First, a *broadcast barrier* is invoked before the parallel 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 invoked after the loop body to ensure all worker threads have completed before the master can continue.

In previous work, we developed compiler algorithms for barrier elimination based on the observation that synchronization between a pair of processors is only necessary if they communicate shared data [25]. If *data dependence analysis* can show two adjacent loop nests access disjoint sets of data, then the barrier separating them may be eliminated, as in Figure 2(B). If two loop nests accesses the same data, but *communication analysis* proves no remote data is accessed based on data and computation decomposition information, the barrier may also be eliminated, as in Figure 2(C). Finally, if communication analysis identifies simple interprocessor sharing patterns, the barrier may be replaced with less expensive forms of synchronization. In particular, if data is only shared between neighboring processors, the barrier may be replaced by nearest-neighbor synchronization, as shown in Figure 2(D).

```
for each pair of parallel loop nests
    if loop bounds are identical
        for each dependence crossing loop nests
            examine array reference pair at endpoints
            find subscript pairs containing parallel loop index
            if any subscript pair is identical
                no barrier needed
            else if any subscript pair differs by constant
                sync_neighbor needed
            else
                barrier needed
        if barrier needed for any dependence
            insert barrier
        else if sync_neighbor needed for any dependence
            insert nearest-neighbor synchronization
        else
            no barrier needed between loop nests
    else
        insert barrier
```

Figure 3: Local Subscript Analysis Algorithm

2.2 Novel Techniques

2.2.1 Communication Analysis Using Local Subscripts

Previous communication analysis relied on compile-time information on the data and computation decomposition selected for a program to precisely determine whether interprocessor communication takes place [25]. We find that an alternative communication analysis technique based on *local subscript analysis* can also yield good results, but with less complex analysis. The Local subscript analysis algorithm is shown in Figure 3.

We assume dependence analysis has already eliminated all array reference pairs between loop nests proven to access disjoint memory locations. Local subscript analysis relies on the compiler selecting a consistent chunk partitioning of parallel loop iterations. Consistent iteration partitioning of loops with identical loop bounds will then always assign the same loop iterations to each processor. If any subscript pair containing parallel loop index is identical, each processor will only access local data. If the subscripts differ by only a constant, then each processor will access remote data a constant number of processors away. Since constant differences in subscripts are usually small, processors will end up accessing remote data on neighboring processors, allowing barriers to be replaced by nearest-neighbor synchronization.

Local subscript analysis is designed to complement full communication analysis. It is less precise, but can be computed efficiently because it only relies on local symbolic information. Local subscript analysis can thus quickly eliminate simple array reference pairs that do not require synchronization, reducing the number of dependences that must be analyzed with communication analysis. Local subscript analysis can also be applied in some cases where standard communication analysis fail, since it relies only on local program information. For instance, in Figure 4(A), local subscript analysis does not need to fully analyze $f_1()$ or $f_2()$ before eliminating the barrier. In comparison, communication analysis must calculate what data is communicated, and can be disabled by a single complex subscript.

2.2.2 Exploiting Lazy Release Consistency

A second enhancement to our compile-time barrier elimination algorithm is made possible by the underlying semantics of lazy-release-consistency software DSMs. In a traditional shared-

```
{ Local Subscript Analysis }
DOALL J = 1,N
  DO I = 1,N
    A(f1(I),f2(J)) =
  ENDDO
ENDDO
DOALL J = 1,N
  DO I = 1,N
    B(...) = A(f1(I),f2(J))
  ENDDO
ENDDO
⇓
broadcast
DO J = LB1,UB1
  DO I = 1,N
    A(f1(I),f2(J)) =
  ENDDO
ENDDO
DO J = LB1,UB1
  DO I = 1,N
    B(...) = A(f1(I),f2(J))
  ENDDO
ENDDO
barrier

(A)
```

```
{ Lazy Release Consistency }
DO TIME =
  DOALL J = 1,N
    DO I = 1,N
      A(I,J) = B(I,J-1)+B(I,J+1)
    ENDDO
  ENDDO
  DOALL J = 1,N
    DO I = 1,N
      B(I,J) = A(I,J)
    ENDDO
  ENDDO
ENDDO
⇓
broadcast
DO TIME =
  DO J = LB1,UB1
    DO I = 1,N
      A(I,J) = B(I,J-1)+B(I,J+1)
    ENDDO
  ENDDO
  { ANTI-DEPENDENCE ONLY,
    NO BARRIER NEEDED }
  DO J = LB1,UB1
    DO I = 1,N
      B(I,J) = A(I,J)
    ENDDO
  ENDDO
  sync_with_neighbors
ENDDO
barrier

(B)
```

Figure 4: Advanced Optimization Examples

memory model, synchronization is needed between two loop nests if two processors access shared data, with at least one of the processors performing a write to the shared data. However, in a lazy-release-consistency software DSM, if the shared reference is a read in the first loop and a write in the second loop, no synchronization is needed because writes from the second loop will not become visible to the read in the first loop until synchronization is encountered. In other words, anti-dependences (write-after-read) do not need to be synchronized.

To see how the compiler can use this property, consider the example shown in Figure 4(b). The first loop nest reads nonlocal values of B which are defined in the second loop nest. The cross-processor dependence caused by B is thus a loop-independent anti-dependence. Normally, synchronization is needed to ensure the old values of B are read before the new values of B are written. However, with lazy release consistency the software DSM guarantees that new values of B on another processor will not be made visible until the two processors synchronize. Since there are no other loop-independent dependences between the two loop nests, synchronization between them is not required.

A cross-processor true/flow dependence (read-after-write) exists which does need synchronization. It is the dependence between definitions of B in the second loop nest and reads of nonlocal values of B in the first loop nest. This dependence is carried by the outer TIME loop, since the endpoints of the dependence occur on different iterations of the TIME loop. The compiler normally inserts a barrier as the last statement of the TIME loop, but local subscript analysis can show only nearest-neighbor synchronization is needed.

The one case where synchronization is needed for anti-dependences is when the processor performing a read does not yet possess a copy of the shared data, since it may retrieve a copy of the data with the new values. For scientific computations where iterative computations are the rule, this is rarely the case. Our implementation of nearest-neighbor synchronization solves this problem by invoking a global barrier the first time it is invoked at each location in the program. Since anti-dependences may be ignored, the algorithm for inserting barrier synchronization becomes similar to the algorithm for *message vectorization* [13]. The level of the deepest true/flow cross-processor dependence becomes the point where synchronization must be inserted to prevent data races. Synchronization at lower loop levels is not needed.

2.2.3 Customized Nearest-Neighbor Synchronization

At some barriers, the compiler can detect communication only takes place between neighboring processors [25]. To take advantage of this information, we implemented a customized routine for nearest-neighbor synchronization (where each processor has either zero, one, or two neighbors) directly in CVM. The routine sends a single message to each neighboring processor upon arrival, and continues as soon as messages are received from all neighboring processors. In comparison, for normal global barriers all processors send a single message to the barrier master, which broadcasts a reply once all processors have checked in.

The customized nearest-neighbor synchronization has several advantages over standard global barriers. The most important is that nearest-neighbor synchronization allows at least some of the induced load imbalance to smooth out before it delays all processors. However, this benefit usually occurs only if there are multiple nearest-neighbor synchronization events invoked in sequence. If nearest-neighbor synchronization and global barriers are executed in alternating sequence, opportunities to smooth out load imbalance are lessened. Second, the serial bottleneck of the barrier master is avoided. This is not a large advantage for the size of the system that we are currently evaluating, but should be significant for larger systems. Finally, common messages can be used to carry both synchronization and data, because both flow only between neighbors.

2.3 Experimental Results

2.3.1 Applications

We evaluated the performance of our compiler/software DSM interface with five programs shown 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. `expl`, and `redblack` 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. `swm` and `tomcatv` are programs from the SPEC benchmark suite containing a mixture of stencils and reductions. We used the version of `tomcatv` from APR whose arrays have been transposed to improve data locality.

Name	Description	Problem Sizes *Granularity (secs)* small	large
expl	Explicit Hydrodynamics (Livermore 18)	256^2 *0.06*	512^2 *0.34*
jacobi	Jacobi Iteration w/Convergence Test	512^2 *0.06*	1024^2 *0.91*
redblack	Red-Black Successive-Over-Relax.	512^2 *0.01*	1024^2 *0.14*
swm	Shallow Water Model (SPEC)	512^2 *0.10*	750^2 *0.20*
tomcatv	Vector Mesh Generation (SPEC)	256^2 *0.04*	512^2 *0.15*

Table 1: Applications

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. Optimized versions of each program were automatically generated by the SUIF compiler. The compiler analyzed but was unable to apply synchronization optimizations to four other programs, indicating that not all programs may benefit from the techniques presented in this paper.

2.3.2 Experimental Environment

We evaluated our optimizations on an IBM SP-2 with 66MHz RS/6000 Power2 processors operating AIX 4.1. Nodes are connected by a 120 Mbit/sec bi-directional Omega switch capable of a sustained bandwidth of approximately 40 Mbytes per second. 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.

In our experiments, CVM [15] applications written in Fortran 77 were automatically parallelized by the Stanford SUIF parallelizing compiler version 1.1.2 [11], with close to 100% of the computation in parallel regions. A simple chunk 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 2.7.2 with the -O2 flag, then linked with the SUIF run-time system and the CVM libraries to produce executable code on the IBM SP-2. Customized support for reductions and the *flush update* protocol were used to improve overall performance [17].

2.3.3 Effectiveness of Compiler Synchronization Optimizations

First, we examine the effectiveness of compiler algorithms in eliminating synchronization. Table 2 displays the number of barriers found in each program at compile time, and the percentage eliminated by different levels of optimization. Table 3 presents the same information for barriers executed dynamically at run time for each application. The first two columns for "barriers" indicate the number of barriers executed in the original program and the percentage reduction by merging *doall*s into the same parallel region. The remaining columns show the percentage of barriers eliminated or replaced by nearest-neighbor synchronization for different levels of optimization. The compiler optimization levels are as follows: "dep" eliminates barriers in parallel regions based on data dependences, "comm" performs communication analysis using local subscript analysis to eliminate barriers, "lazy" eliminates barriers guarding only anti-dependences. Communication analysis may also replace barriers with nearest-neighbor synchronization, Optimizations are cumulative.

We see from both tables that the compiler is effective at eliminating parallel loops and barriers encountered during compilation, with roughly similar benefits for the number of parallel loops and barriers actually executed by the application at run time. Examining the run-time measurements in Table 3, we find the compiler is quite successful in discovering parallel loops which may be merged into a single parallel region, eliminating on average 30% of barriers executed which is equivalent to 60% elimination of *doall* invocations. Dependence analysis alone is only able to eliminate barriers in two programs, `redblack` and `swm`, but the improvement in `redblack` is significant. Communication analysis can eliminate barriers in `tomcatv` and replace barriers in `expl`, `jacobi`, and `redblack`. Detecting barriers guarding only anti-dependences, the compiler can eliminate more barriers outright in `expl` and `jacobi`. The number of replaced barriers goes down in `expl` and `jacobi`, since the compiler can prove some nearest-neighbor barriers guard only anti-dependences. Applying all optimizations, on average 53% of all barrier executions are eliminated in these five programs, with 6% of barriers replaced by nearest-neighbor synchronization.

Program	orig	Barriers in program % eliminated [% replaced]					
	num	merge	dep	comm		lazy	
expl	6	33	33	33	33	50	17
jacobi	4	25	25	25	25	50	–
redblack	8	38	63	63	13	63	13
swm	22	18	27	27	–	27	–
tomcatv	16	31	31	62	–	62	–
Average	11.2	29	36	42	14	50	6.0

Table 2: Static Measurement of Synchronization Optimizations

Program	orig	Barriers executed by program % eliminated [% replaced]					
	num	merge	dep	comm		lazy	
expl	120	33	33	33	33	50	17
jacobi	80	25	25	25	25	50	–
redblack	160	38	63	63	13	63	13
swm	530	17	33	33	–	33	–
tomcatv	280	36	36	71	–	71	–
Average	234	30	38	45	14	53	6.0

Table 3: Dynamic Measurement of Synchronization Opts

2.3.4 Impact of Compiler Optimizations on Synchronization Overhead

In order to evaluate synchronization optimizations in more detail, we instrumented CVM to directly measure barrier overhead (time spent executing barrier code), load imbalance (idle time waiting for completion of current parallel task) and sequential wait(idle time waiting for next parallel task). We found the actual time spent in barrier routines to be small(refer to figure 1). Instead, most of the overhead was caused by sequential wait and load imbalance. Table 4 shows sequential wait and load imbalance as percentage of overall execution time. The measurements are for 16 processor SP-2 runs with small and large data sets. Average sequential wait and load imbalance are 14% and 27% of execution time for large data sets and 23% and 31% for small data sets respectively. Load imbalance takes up less percentage of execution time with large data sets, because the larger amount of data on individual processors gives the greater chance to smooth out load imbalance. We

		% Exec time (unopt)		% Decrease in seq_wait + imbalance		
		seq_wait	imbal.	dep	comm	lazy
expl	sm	20	15	37	36	48
	lg	6.5	8.9	22	26	31
jacobi	sm	27	24	31	37	52
	lg	14	15	27	36	39
redblack	sm	36	36	67	72	72
	lg	29	32	66	69	69
swm	sm	15	32	17	17	17
	lg	12	34	16	16	17
tomcatv	sm	18	47	18	23	25
	lg	10	46	7.7	12	14
Average	sm	23	31	34	37	43
	lg	14	27	28	32	34

Table 4: Effect of Optimizations on Load Imbalance and Sequential Wait (16 Processors)

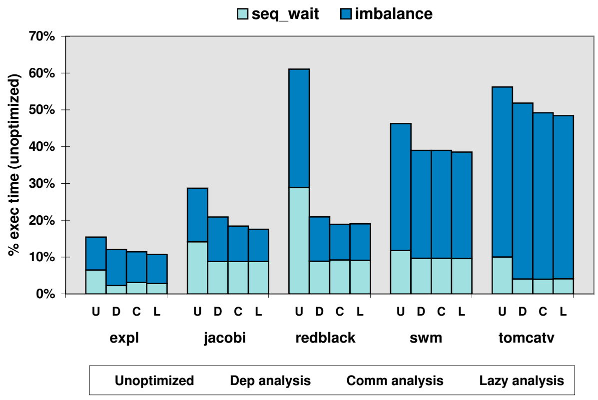

Figure 5: Effect of Optimizations on Load Imbalance (16 Processors)

also measured the impact of synchronization optimizations on reducing load imbalance. Table 4 displays the percentage reduction in idle time after applying different levels of optimization. Figure 5 graphically presents the same information for the large data sets. We see that optimizations can significantly reduce sequential wait and load imbalance for some of the applications studied (72% decrease for `redblack` with small data set). Average sequential wait and load imbalance decrease by 28%, 32%, and 34% with large data sets and by 34%, 37%, and 43% with small data sets for three optimization levels, respectively.

2.3.5 Impact of Compiler Optimizations on Program Performance

Figure 6 displays the impact of synchronization optimizations on application performance on the SP-2, for both small and large data sets. For each graph, the Y-axis measures improvement over unoptimized programs, the X-axis presents three optimized versions of each program for both 8 and 16 processor runs: "dep analysis" merges parallel regions and eliminates barriers based on data dependences, "comm analysis" uses local subscript analysis to eliminate barriers or replace them with nearest-neighbor synchronization, "lazy analysis" eliminates barriers guarding only anti-dependences. Except for `redblack` and `swm`, performance for "dep analysis" is the same as that for simply merging adjacent parallel loops into the same parallel region. Optimizations are cumulative.

Performance for applications cover a broad range. For 16 processor runs with large data sets, average improvement from synchronization optimizations is 14% for dependence analysis, 17%

		Speedup			
		unopt	dep	comm	lazy
expl	sm	5.7	6.6	6.9	7.1
	lg	13	13	14	14
jacobi	sm	6.0	6.8	7.3	8.2
	lg	12	13	13	13
redblack	sm	1.3	2.7	3.0	3.1
	lg	4.0	6.9	7.2	7.2
swm	sm	1.4	1.5	1.5	1.5
	lg	2.2	2.4	2.4	2.4
tomcatv	sm	1.3	1.5	1.7	1.7
	lg	3.1	3.3	3.6	3.6
Average	sm	3.1	3.8	4.1	4.3
	lg	6.7	7.7	7.9	8.1

Table 5: Impact of Barrier Elimination (16 Processors)

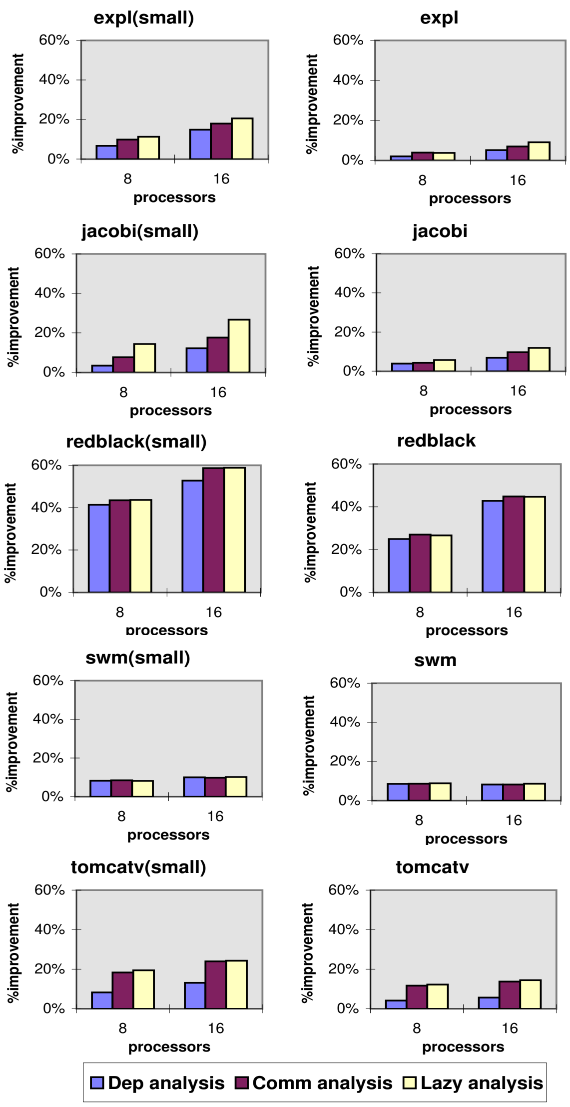

Figure 6: Impact of Barrier Elimination

for communication analysis, and 18% when also eliminating barriers using lazy release consistency. As expected, optimizations have greater impact for smaller data sets and more processors, since synchronization overhead is more significant. For small data set runs on 16 processors, average improvements increase to 21%, 26%, and 28% respectively. For 8 processor runs improvements from synchronization optimizations drop to 9%, 11%, and 11% with large data sets and 14%, 17%, and 18% with small data sets for the three optimization levels, respectively. Table 5 also displays speedups due to optimizations in detail.

2.3.6 Impact of Nearest-Neighbor Synchronization on Performance

Using communication analysis, SUIF was able to replace two barriers with nearest-neighbor synchronization in `expl`, and one barrier each in `jacobi` and `redblack`. The performance improvements due to replacing barriers are shown in Table 6. As with all synchronization optimizations, impact is heightened for smaller data sizes. Performance improvements are due to a combination of better load balancing and fewer messages; the amount of data communication that is piggybacked on synchronization messages tripled for some applications when using nearest-neighbor barriers. Nonetheless, the overall impact on performance is limited. The primary reason is that many of the barrier synchronizations that are prime candidates to be replaced by nearest-neighbor synchronization can be eliminated instead. However, nearest-neighbor synchronization may prove more important when using more than 16 processors.

Execution time (dep vs. comm)	% Improvement small	large
expl	3.5	2.4
jacobi	5.5	2.5
redblack	9.3	5.0
swm	–	–
tomcatv	–	–
Average	3.7	2.0

Table 6: Impact of Nearest-Neighbor Barriers (16 Processors)

3 Data Layout Transformations

In addition to synchronization optimizations, we are also investigating *data layout* transformations for both uniprocessor and multiprocessor architectures. Because of the increasing gap between processor and memory speeds, applications that do not use caches effectively spend most of their time waiting for memory accesses. Due to hardware constraints, caches have *set associativity*, where memory addresses can only be mapped to one of k locations in a k-way associative cache. Limited set associativity can reduce the amount of reuse programs exploit. *Conflict misses* may occur when too many data items map to the same set of cache locations, causing cache lines to be flushed from cache before they may be reused, despite sufficient capacity in the overall cache. Conflict misses have been found to be a significant source of poor performance in scientific programs, particularly within loop nests [18].

3.1 Eliminating Conflict Misses

Compiler transformations can be very effective in eliminating conflict misses for scientific programs with regular access patterns. We evaluate two compiler transformations to eliminate conflict misses: 1) inter-variable padding (modify variable base address), 2) intra-variable padding (increase array dimension size) [2, 24]. Unlike compiler transformations that restructure the computation performed by the program, these two techniques modify the program's *data layout*.

Two motivating examples of these transformations are shown in Figures 7 and 8. In Figure 7, unit-stride references to `B(i)` and `C(i)` provide spatial locality, leading to cache reuse. However, if `B` and `C` are separated by a multiple of the cache size and the cache is direct mapped, every access to `C(i)` will map to the same cache line as the previous `B(i)` access and vice versa. As a result, every reference will produce a conflict miss. To regain the benefits of spatial reuse, inter-variable padding may be applied as shown in Figure 7.

In Figure 8, the stencil computation exhibits much spatial and temporal reuse between the references to `B`. However, if the column size of `B(i)` is a multiple of the cache size, conflict misses will result, eliminating reuse between references to `B(i).` To regain spatial reuse, intra-variable padding applied to the columns of `B` can change its internal layout so that columns no longer map to conflicting locations on the cache, as shown in Figure 8.

Both of these examples show instances of *severe* conflict misses, when conflict misses occur on every loop iteration. In the remainder of this section, we evaluate several heuristics for applying data layout transformations to eliminate severe conflict misses.

```
real S, B(N), C(N)
do i = 1,N
  S = S + B(i)*C(i)

--> real B(N), DUMMY(PAD), C(N)
```

Figure 7: Inter-Variable Padding (Change Base Address)

```
real A(N,N), B(N,N)
do j = 1,N
  do i = 1,N
    A(i,j) = 0.25 * (B(i+1,j)+B(i-1,j)
                    +B(i,j+1)+B(i,j-1))

-->  real A(N,N), B(N+PAD,N)
```

Figure 8: Intra-Variable Padding (Change Column Dimension)

3.2 Analysis-Free Padding

We begin with heuristics which examine only variable base addresses and array dimension sizes. These analysis-free heuristics are very simple and do not require compile-time program analysis. In particular, the inter-variable padding heuristic is a reasonable link-time optimization heuristic, because only variable base address information is examined and modified.

3.2.1 Inter-variable Padding Heuristic

We first divide arrays into groups based on array size, motivated by the assumption that most severe conflicts arise from interference between references to variables of the same size. While separation by one cache line will ensure that references of the form `B(i)` and `C(i)` do not conflict, a larger separation may be required for references such as `B(i)` and `C(i+2)`. A small number of cache lines D_L can be adopted as a minimum distance of separation, with the intention of avoiding conflicts between the most common forms of references.

Assuming cache size C_S and cache line size L_S, we assign array base addresses to a maximum of $\frac{C_S}{D_L L_S}$ locations, separated by D_L cache lines. Based on empirical studies [22], our implementation currently separates array base addresses by four cache lines (D_L = 4). An alternative heuristic which separates arrays as far as possible in the cache was shown to be less effective in eliminating conflict misses [22].

3.2.2 Intra-variable Padding Heuristic

A possible intra-variable padding heuristic is to pad the dimensions of each array so that the size of subarrays comprising lower dimensions are far enough from any cache size multiple. In the simplest case, array columns would be padded when neighboring columns are mapped to the same location in cache. Empirical studies [22] indicate this heuristic misses many profitable opportunities for intra-variable padding. As a result, our implementation currently pads all array columns by four elements unconditionally when safe. It should be clear that this heuristic is not practical; however, our experiments demonstrate it is quite effective in identifying many programs which stand to benefit from intra-variable padding.

3.3 Analysis-Based Padding

For more sophisticated padding heuristics, additional compiler analysis of array references is required. First, we find *uniformly generated* references described by Gannon *et al.* [9]. We then compute the *conflict distance*, the difference between the memory addresses of two array references modulo the cache size C_S. If on each loop iteration the conflict distance is less than the cache line size L_S, severe conflict misses can result. Observe that making the conflict distance L_S or greater is a sufficient (though not necessary) condition for eliminating severe conflicts; we use this to guide our data layout optimizations. Our techniques for computing conflict distances is a special simplified case of *cache miss equations* [10].

3.3.1 Inter-variable Padding Heuristic

Central to our technique for eliminating severe conflict misses is considering pairs of arrays references with constant conflict distances on each loop iteration. We find that such pairs are commonly uniformly generated references [9] extended for conforming arrays, arrays that have equal dimension sizes in all but their highest dimension and possess equal-sized elements. A pair of uniformly generated references to d-dimensional conforming arrays `A` and `B` has the form $A(i_1 + r_1, i_2 + r_2, ..., i_d + r_d)$ and $B(i_1+s_1, i_2+s_2, ..., i_d+s_d)$, where each i_j is an integer variable or the value 0, and each r_j and s_j is some constant integer value. Uniformly generated references may include references nested in for loops with indices i_j, references to 1-dimensional arrays of arbitrary size, and references with integer subscripts (in which case i_j reduces to 0).

When we calculate the conflict distance on a given loop iteration by subtracting the two memory locations accessed by uniformly generated references, we obtain an expression in which all i_j terms cancel:

$$|BaseAddr(A) - BaseAddr(B) + \sum_{j=1}^{d}([r_j - s_j]S_e \prod_{k=1}^{j-1} S_k)| \quad (1)$$

In this expression, $BaseAddr(A)$ and $BaseAddr(B)$ are the base addresses of `A` and `B`, S_e is the common element size, and S_k is the comment size of dimension k. A lower bound of 0 is assumed in each dimension, but other lower bounds $l_{A,j}$ and $l_{B,j}$ may be accounted for by adding $l_{B,j} - l_{A,j}$ to the factor $[r_j - s_j]$. Taking the modulo of the cache size C_S then yields the conflict distance.

We wish to identify all pairs of uniformly generated references which may conflict. Our algorithm therefore analyzes each loop to find such references. Since references R and S being uniform and references R and T being uniform implies S and T are uniform as well, pairs of uniformly generated references in a loop can be efficiently represented as *groups* of uniformly generated references. Each reference in a group is uniform with respect to all other references in the group. Our initial analysis is thus is to compute for every loop the the set of maximal groups of uniformly generated references.

We apply inter-variable padding as follows. The objective is to assign each array a base address so that no two uniformly-generated references in a group conflict. We accomplish this in a simple, greedy fashion. All array base addresses are initially undefined. Then, each array in turn is assigned a base address as follows. First, the next available address is attempted. If this address leads to a conflict with any other array with a defined base address, the address is incremented by the cache line size L_S, and the process repeats. The test for a conflict between two arrays involves instantiating expression 1 with the r and s values from all pairs referencing the two arrays in each of the analyzed groups, and the S_e, S_j, and BaseAddr values imposed by the arrays under consideration. If the conflict distance obtained is less than L_S, then we assume the addresses conflict.

A base address needs to be incremented at most $N = C_S/L_S$ or more times, since conflicts for the first N values implies conflicts for any remaining values obtained by adding L_S. If this case happens, the algorithm has failed to find a satisfactory address and assigns the first, unpadded address attempted. In our experiments the compiler can always find a non-conflicting base address.

3.3.2 Intra-variable Padding Heuristic

Intra-array padding aims to modify the distances between references to the *same* array in order to eliminate conflicts. The notion of uniformly generated references is again useful in identifying common forms of array references accessing locations of a fixed distance. By the definition given in the last section, two references to the same array are uniformly generated as long as they are of the form $A(i_1+r_1, i_2+r_2, ..., i_d+r_d)$, $A(i_1+s_1, i_2+s_2, ..., i_d+s_d)$, again where each i_j is an unmodified variable or 0, and each r_j and s_j is an arbitrary integer. All other conditions are always satisfied since the two references access the same array. Expression 1 in this special case reduces to

$$|\sum_{j=1}^{d}([r_j - s_j]S_e \prod_{k=1}^{j-1} S_k)| \quad (2)$$

since the base addresses (and any array lower bounds) cancel each other out. Stated in terms of expression 2, intra-array padding modifies the S_j values for each array as needed to produce non-conflicting distances between all uniformly generated references.

We apply intra-variable padding as follows. First, each loop is analyzed to compute maximal groups of uniformly generated references to the same array. Arrays passed to procedures are not padded, since no interprocedural analysis is performed. Second, the compiler instantiates expression 2 with the r and s values of all reference pairs from each group containing references to the array, and the S_e and S_j values imposed by current array layout. If the resulting distance conflicts, a pad of one element is attempted on all of the lower dimensions until no conflicts are found. If the pad size fails, it is incremented by one, and the reanalyzed. An upper bound on the padding size is imposed to ensure termination. In our experiments, for all programs encountered no pad greater than two elements was required.

3.4 Prototype SUIF Implementation

To evaluate the impact of compiler analysis on padding heuristics, each of the four transformations (analysis-free/analysis-based intra/inter-variable padding) were implemented as passes in SUIF. When applying both intra/inter-variable padding in combination, we perform intra-variable padding before inter-variable padding.

Before performing padding, a number of transformations are applied to each program in a preprocessing phase. Preprocessing begins with constant propagation, which can have the effect of transforming array subscripts involving the addition of multiple variables to the form required of uniformly generated references. In this way, constant propagation can improve the precision of the analysis.

The remaining transformations are performed to give the optimizer compile-time control over variable base addresses. First, local array and structure variables are promoted into the global scope. Formal parameters to functions cannot and do not need to be promoted, since they represent variables declared elsewhere. Array and structure global variables are then made into fields of `suifmem`, a large structured variable (or common block in Fortran). Structures/common blocks which are themselves fields of `suifmem` are recursively split so that individual variables may be moved about as necessary. Preprocessing thus results in a single global variable containing all of the variables to be optimized, as shown in Figure 9. Optimizing passes may now modify the base addresses of variables by reordering fields in the `suifmem` structure and inserting pad variables.

3.5 Experimental Evaluation

Each transformation was applied to a number of scientific kernels and applications taken from the SPEC92 and NAS benchmarks

program	arrays	intra-array padding				inter-array padding	array references			
		safe	need	padded	largest pad	lines skipped	num	% uniform	% not uniform	% not global
adi32	6	100	50	50	2	2	39	100		
dgefa256	2	100					20	90	10	
dot256	2	100				1	4	100		
erle64	23	30	9	9	1	1	274	70		30
expl512	9	100				14	61	100		
irr500k	4						24	29		71
jacobi512	2	100				2	10	100		
linpackd	6	17					262	45		55
mult300	3	100					7	100		
rb512	1	100					25	100		
shal512	15	93	40	40	1	8	195	100		
doduc	91	70				5	2814	27	6	67
fpppp	23	70					1439	74	3	23
hydro2d	9	100					717	2		98
ora	23	70					374	16	13	71
su2cor	117	61				19	1428	55	14	31
swm256	14	100					193	100		
tomcatv	7	100				1	99	84	16	
appbt	42	62					3579	62		38
applu	34	50					1504	97		3
appsp	41	56	2	2	1	15	1669	73		27
buk	5	20				1	46	24	7	70
embar	3	33					10	40	50	10
fftpde	7					2	135	19	4	78

Table 7: Padding Statistics for Optimizations (16K direct-mapped cache, 32B lines)

```
int A[100];       --> struct _globmem {
foo() {                 ...
  int B[100];           int A[100];
  for (i)               int B[100];
    A[i] = B[i];        ...
}                     } suifmem;

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

Figure 9: Globalization in preprocessing stage

suites. The original, preprocessed, and optimized versions of each program were both timed and simulated. Timings were made on 275MHz DEC Alpha 20064 processors with 16KB direct-mapped primary caches and 32B cache lines. Application cache behavior was also simulated for the same cache configuration using the ATOM binary rewriting tool. In addition, various measures of the progress of the analysis-based transformation were compiled. The resulting observations demonstrate the potential effectiveness of our transformations.

3.5.1 Padding Statistics

Our experiments indicate that for programs tested, intra and inter-variable pads were easily computed by our heuristics. Table 7 details compile-time statistics on the success of our padding heuristics. Intra-variable padding provided the data in columns *safe* through *largest pad*. These columns contain for each program the percentage of arrays deemed *safe* to pad, the percentage of safe arrays which *needed* padding, and the percentage of arrays consequently *padded* (hence *safe* $\geq$ *need* $\geq$ *padded*). *Largest pad* indicates the largest pad required. Results show analysis-based heuristics decided only four programs (adi32, erle64, shal512, appsp) required intra-variable padding. All arrays requiring padding were legally padded, and small pad sizes ($\leq$ two elements) were sufficient for these programs.

The next column, *lines skipped*, relates the total number of cache line-sized pads inserted between variables during analysis-based inter-variable padding (after intra-variable padding). We see about half of the programs required padding, usually only by a few cache lines. Though our algorithm could conceivably skip many more lines by adding L_S as many as $\frac{C_S}{L_S} - 1$ times to each candidate base address, in each test the combined total for all variables was always less than 20.

The final columns categorize the array references analyzed in each program. The four columns under *array references* indicate the number of array references (*num*), the percentage of array references which enter some group (*uniform*), the percentage of global array references which by their form cannot be unformly generated *not uniform*, and the percentage of array references which aren't global *not global*. (Thus, *uniform* + *not uniform* + *not global* = *100*). Only *uniform* references are analyzed; *not uniform* and *not global* references, though common in a number of programs, are ignored. Results show most references were uniformly generated, though many belonged to procedure parameters which could not be transformed.

3.5.2 Cache Miss Rates

Table 8 presents the impact of optimizations on simulated cache miss rates. For each program, miss rates are given for the original version and a number of versions transformed by the analysis-free and analysis-based heuristics. The best improvement compared to the original program is presented both as the drop in cache miss rate (*best improve*) and as the percent decrease in miss rate (*best improve %*). Overall, data layout optimizations are very effective in eliminating cache misses in many programs. Cache miss rates are reduced by 5% or more in eleven of the 24 programs tested, and by 20% or more in six programs. Degradations in performance do occur. No optimized version outperforms the original in the cases of doduc, fpppp, hydro2d, and appbt. Miss rate increases of 0.1% occur in three other programs as well. However, all of these

		analysis-free		analysis-based			
program	orig	intra	intra+inter	intra	intra+inter	best improve	best improve (%)
adi32	59.6	15.6	24.8	11.7	11.7	47.9	80
dgefa256	23.3	23.3	13.8	23.3	23.3	9.5	41
dot256	66.1	16.7	16.7	16.7	16.7	49.4	75
erle64	21.0	17.8	5.2	17.8	4.8	16.2	77
expl512	77.4	13.9	14.0	13.9	13.9	63.5	82
irr500k	3.8	3.8	3.8	3.8	3.8	0.0	0
jacobi512	28.9	8.8	8.8	8.8	8.8	20.1	70
linpackd	10.1	10.1	10.1	10.1	10.1	0.0	0
mult300	10.4	10.4	10.1	10.4	10.4	0.3	3
rb512	17.8	17.8	17.9	17.8	17.8	0.0	0
shal512	30.7	5.0	5.1	5.2	5.3	25.7	84
doduc	4.6	5.6	5.0	5.3	5.3	-0.4	-8
fpppp	3.6	4.0	5.5	3.8	3.8	-0.2	-5
hydro2d	13.2	13.9	13.5	13.5	13.5	-0.3	-1
ora	0.0	0.0	0.0	0.0	0.0	0.0	0
su2cor	39.9	19.6	19.7	19.6	19.6	20.3	51
swm256	3.9	3.9	4.0	3.9	3.9	0.0	0
tomcatv	18.6	7.7	9.1	7.7	7.7	10.9	59
appbt	3.6	4.3	4.4	3.9	3.9	-0.3	-7
applu	4.4	4.3	4.4	4.5	4.5	0.1	2
appsp	27.3	16.2	16.3	15.0	15.0	12.3	45
buk	8.3	1.5	1.5	1.5	1.5	6.8	82
embar	0.7	0.7	0.7	0.7	0.7	0.0	0
fftpde	9.2	9.0	9.0	9.0	9.0	0.2	2
Average	20.3	9.7	9.3	9.5	9.0	11.8	31

Table 8: Impact of Optimizations on Cache Miss Rates (16K direct-mapped cache, 32B lines)

increases are relatively mild. In no programs do miss rates rise by more than 1%.

In comparing the analysis-free approach with the analysis-based approach, we find that while each may perform better in certain cases, over the entire range of programs tested, the analysis-based approach performs somewhat better. The row labeled *Average* reveals the magnitude of these differences. Programs undergoing intra-variable padding without analysis average a 9.7% miss rate versus 9.5% under the analysis-based transformation. A similar 9.3% to 9.0% ratio results in the case of composed intra+inter-variable padding. Since the analysis-free intra-variable transformation pads unconditionally, it inevitably degrades performance in a number of programs. The worst example is adi32, for which the miss rate increases by over 9%. The analysis-based intra-variable transformation makes no serious mistakes.

Many such weaknesses in our present implementation for analysis-based padding may stem from the simplicity of its analysis. For instance, references to arrays which are formal parameters are ignored during analysis. Interprocedural analysis will enable the compiler to map these formal parameters to one or more global arrays (or to offsets of global arrays), allowing them to be optimized.

Additional scalar analysis should improve the level of analysis as well. In particular, analysis may stand to benefit considerably by the forward propagation of variable assignments into array subscripts. We illustrate this point with sample code from the SPEC benchmark tomcatv. The nest in Figure 10 contains 8 array references whose subscripts include 6 different variables. However, variable `JP` and `JM` contain the values `J+1` and `J-1` throughout the nest. `IP` and `IM` are related to `I` similarly. When we propagate the assignments of these variables into the array subscripts, all 8 references can be confirmed as uniformly generated and enter the same group during analysis. Without forward propagation, analysis fails to detect this relationship, producing 4 groups with 2 references each.

We should note that in our tests, we used a version of tomcatv

```
DO J = J1P,J2M
  JP = J+1
  JM = J-1
  ...
  DO I = I1P,I2M
    IP = I+1
    IM = I-1
    XX = X(IP,J)-X(IM,J)
    YX = Y(IP,J)-Y(IM,J)
    XY = X(I,JP)-X(I,JM)
    YY = Y(I,JP)-Y(I,JM)
    ...
  CONTINUE
 CONTINUE
```

Figure 10: Tomcatv nest requiring forward propagation

with the forward propagation described above. When applied the unmodified tomcatv, our analysis-based transformations obtained miss rates of 13.7%. Forward propagation reduced these values to 7.7%. These results indicate that data layout optimizations based on uniformly generated references will be most effective only in conjunction with standard forms of scalar analysis.

3.5.3 Execution Time

Versions for programs exhibiting a significant (5% or more) reduction in miss rates were timed on DEC Alphas, as shown in Table 9. Results demonstrate that observed miss rate reductions translate into improvements in performance. All execution times decrease by 10% or more. For the programs given in this table, the analysis-based transformations again do slightly better overall. However, with respect to the *intra+inter* averages presented, the analysis-based heuristic owes a significant share of its advantage

program	orig	no analysis		analysis		best improve (%)
		intra	intra+inter	intra	intra+inter	
adi32	7.9	6.5	7.5	6.5	6.4	19
dgefa256	0.9	0.8	0.7	0.8	0.8	22
dot256	10.1	8.5	8.5	8.5	8.5	16
erle64	1.5	1.5	1.4	1.5	1.3	13
expl512	32.3	18.8	19.1	18.8	18.8	42
jacobi512	11.7	5.1	4.4	5.1	5.1	62
shal512	43.4	30.8	30.9	30.7	29.9	31
su2cor	158.0	141.6	141.3	141.1	140.9	11
tomcatv	33.0	29.0	29.7	29.2	29.2	12
appsp	17.7	16.1	16.1	16.0	16.0	10
buk	2.0	1.8	1.8	1.8	1.8	10
Average	29.0	23.7	23.8	23.6	23.5	23

Table 9: Impact of Optimizations on Execution Times for DEC Alpha (in seconds)

to the degradation experienced in the version of adi32 transformed by its analysis-free counterpart.

3.5.4 Discussion

In general, we found small inter-variable pads were sufficient to eliminate the worst of the severe conflict misses. Both the analysis-free and analysis-based heuristics were successful, though the analysis-based heuristic is more accurate for adi32. Only two programs (dgefa256, erle64) benefit from intra-variable padding. The analysis-based heuristic identified the need for intra-variable padding for erle64, but fails for dgefa256. The analysis-free heuristic of padding all array columns degrades the performance of adi32.

The advantage of analysis-based heuristics thus appears to be the ability to usually detecting profitable opportunities for padding, while avoiding padding decisions which degrade performance. Previous studies indicate padding transformations become more important as the ratio of application data to cache size increases [22], so analysis-based heuristics may become more important for real-world programs. Our results show not all programs were greatly improved, but the impact is significant enough that data layout optimizations should be considered an important tool for optimizing compilers.

4 Related Work

4.1 Synchronization Optimizations

Researchers compiling for fine-grain data-parallel languages sought to eliminate barriers following each expression evaluation [12, 20]. Simple data dependence analysis can be used to reduce barrier synchronization by orders of magnitude, greatly improving performance. Eliminating barriers in compiler-parallelized codes is more difficult. In previous work [25] we presented techniques to eliminate or lessen synchronization based on communication analysis used by distributed-memory compilers to calculate explicit communication [13]. O'Boyle and Bodin [19] present techniques similar to local subscript analysis to to identify data dependences that cross processor boundaries. Stoher and O'Boyle [23] extend this work present an optimal algorithm for eliminating barrier synchronization in perfectly nested loops.

Recently researchers have investigated optimizations when compiling for software DSMs. Dwarkadas *et al.* applied compiler analysis to explicitly parallel programs to improve their performance on a software DSM [7]. Chandra and Larus evaluated combining the PGI HPF compiler and the Tempest software DSM system [4]. Cox *et al.* conducted an experimental study to evaluate the performance of TreadMarks as a target for the Forge SPF shared-memory compiler from APR [6]. They identify opportunities to eliminate unneeded barrier synchronization; many of their suggestions are implemented in the SUIF/CVM system.

Rajamony and Cox developed a performance debugger for detecting unnecessary synchronization at run-time by instrumenting all loads and stores [21]. In the SPLASH application Water, it was able to detect barriers guarding only anti and output dependences that may be eliminated by applying odd-even renaming. In comparison, SUIF at compile time eliminates many barriers guarding only anti-dependences.

4.2 Data Locality Optimizations

Data locality has been recognized as a significant performance issue for both scalar and parallel architectures. Wolf and Lam provide a concise definition and summary of important types of data locality [26]. Gannon *et al.* introduce the notion of uniformly generated references as a means of discovering group reuse between references to the same array [9]. McKinley and Temam performed a study of loop-nest oriented cache behavior for scientific programs and concluded that conflict misses cause half of all cache misses and most intra-nest misses [18]. Most researchers have concentrated on computation-reordering transformations. Loop permutation and tiling are the primary optimization techniques [3, 9, 26], though loop fission (distribution) and loop fusion have also been found to be helpful [3].

Researchers have previously examined changing data layout in parallel applications, but have generally not studied its effect on sequential applications. Jeremiassen and Eggers automatically eliminate false sharing in explicitly parallel programs [14]. Cierniak and Li examined combining array transpose and loop transformations to improve locality for parallel programs [5]. Amarasinghe *et al.* demonstrated the utility of data layout transformations for parallel applications [1]. They found it to be significant in eliminating adverse cache effects, though specialized optimizations were necessary to reduce computation overhead for modified array subscripts.

Many researchers have also examined the problem of deriving estimates of cache misses in order to help guide data locality optimizations [8, 9, 26]. These models typically can predict only capacity misses because they assume a fully-associative cache. In comparison, Ghosh *et al.* can determine conflict misses by calculating cache miss equations, linear Diophantine equations that summarize each loop's memory behavior by calculating exactly the cache slot to which each reference is mapped [10]. They demonstrate how to use cache miss equations to select array paddings to eliminate conflict misses, and block sizes for tiling. In comparison, we focus only on severe conflicts, using a simplified version of cache miss equations to calculate the *conflict distance* between pairs of uniformly generated array references. Our approach is less precise, but should be more efficient.

5 Conclusions

This paper describes ongoing SUIF compiler projects for high performance architectures at the University of Maryland. First, we are investigating compilation techniques for eliminating barrier synchronization overhead in compiler-parallelized programs running on software distributed-shared-memory (DSM) systems. Second, we are evaluating data layout optimizations, starting with techniques to eliminate cache conflict misses through variable padding. Our optimizations have been implemented in SUIF and tested on a number of programs. Preliminary experimental results are encouraging, but indicate more careful compiler analysis is needed to guide optimization heuristics for both parallel and sequential codes. We intend to continue work in both areas in pursuit of our goal: *efficient* machine-independent programming of both sequential and parallel processors.

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] W. Bolosky, R. Fitzgerald, and M. Scott. Simple but effective techniques for NUMA memory management. In *Proceedings of the Twelfth Symposium on Operating Systems Principles*, Litchfield Park, AZ, December 1989.

[3] S. Carr, K. S. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, San Jose, CA, October 1994.

[4] S. Chandra and J.R. Larus. Optimizing communication in HPF programs for fine-grain distributed shared memory. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Las Vegas, NV, June 1997.

[5] M. Cierniak and W. Li. Unifying data and control transformations for distributed shared-memory machines. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.

[6] A. Cox, S. Dwarkadas, H. Lu, and W. Zwaenepoel. Evaluating the performance of software distributed shared memory as a target for parallelizing compilers. In *Proceedings of the 11th International Parallel Processing Symposium*, Geneva, Switzerland, April 1997.

[7] 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.

[8] J. Ferrante, V. Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing, Fourth International Workshop*, Santa Clara, CA, August 1991. Springer-Verlag.

[9] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5(5):587–616, October 1988.

[10] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: An analytical representation of cache misses. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, Vienna, Austria, July 1997.

[11] 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.

[12] P. Hatcher and M. Quinn. *Data-parallel Programming on MIMD Computers*. The MIT Press, Cambridge, MA, 1991.

[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] T. Jeremiassen and S. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

[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 and C.-W. Tseng. Enhancing software DSM for compiler-parallelized applications. In *Proceedings of the 11th International Parallel Processing Symposium*, Geneva, Switzerland, April 1997.

[18] K. S. McKinley and O. Temam. A quantitative analysis of loop nest locality. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, Boston, MA, October 1996.

[19] M. O'Boyle and F. Bodin. Compiler reduction of synchronization in shared virtual memory systems. In *Proceedings of the 1995 ACM International Conference on Supercomputing*, Barcelona, Spain, July 1995.

[20] M. Philippsen and E. Heinz. Automatic synchronization elimination in synchronous FORALLs. In *Frontiers '95: The 5th Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, February 1995.

[21] R. Rajamony and A.L. Cox. A performance debugger for eliminating excess synchronization in shared-memory parallel programs. In *Proceedings of the Fourth International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, February 1996.

[22] G. Rivera and C.-W. Tseng. Compiler optimizations for eliminated cache conflict misses. Technical Report CS-TR-3819, Dept. of Computer Science, University of Maryland at College Park, July 1997.

[23] E. Stohr and M. O'Boyle. A graph based approach to barrier synchronisation minimisation. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, Vienna, Austria, July 1997.

[24] J. Torrellas, M. Lam, and J. Hennessy. Shared data placement optimizations to reduce multiprocessor cache miss rates. In *Proceedings of the 1990 International Conference on Parallel Processing*, St. Charles, IL, August 1990.

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

[26] M. E. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.