Data Layout Optimizations for High-Performance Architectures

Chau-Wen Tseng
Department of Computer Science
University of Maryland
College Park, Maryland 20742
(301) 405-8010
tseng@cs.umd.edu
http://www.cs.umd.edu/~tseng/

Summary

A key contributor to the success of computational science has been the development of software support which allows users to run their programs efficiently on high-performance architectures (both scalar and parallel) without excessive concern with the details of the underlying hardware architecture. In particular, compilation technology has advanced significantly, to the point where programmers are freed from explicitly managing hardware features such as vector registers, multiple functional units, out-of-order instruction issue, and fine-grain loop parallelism. The increasing disparity between processor and memory speeds has made exploiting reuse of registers and cache ever more important to obtaining high performance. Experience has shown that compilers can automatically restructure the computation performed by a program (e.g., loop permutation, tiling) to improve reuse. Recently, researchers are beginning to discover modifying the layout of data in a program may also be crucial for achieving good performance.

This proposal requests support for research on *data layout* optimizations for both scalar and parallel architectures. These optimizations modify how static and heap allocated variables in a program are laid out in memory, with the goal of improving spatial locality and avoiding adverse memory effects such as conflict misses and false sharing. Techniques include modifying the base addresses of variables, changing fields offsets within structures, group-and-transpose of related data, padding, transposing, and reindexing array dimensions, and modifying heap allocation policies. Most optimizations must be applied at compile time, but link-time and run-time optimizations are also possible. Some optimizations are designed for array-based codes, while others target codes with pointers and complex data structures.

While individual data layout optimizations have been proposed and studied to a limited extent, there is yet no comprehensive examination of their efficacy when applied in conjunction. The main goal of this proposal is to develop heuristics for guiding data layout optimizations, both individually and as part of an overall optimization strategy which includes computation transformations.

The key technical problems this project intends to overcome are 1) developing heuristics needed to effectively guide data layout optimizations for both scalar and parallel machines, 2) determining the precision of compile-time analyses needed to effectively guide these heuristics, 3) combining all data layout optimizations in a single optimization algorithm, 4) integrating data layout and computation-reordering optimizations in the same optimization framework, 5) developing analyses to ensure data layout optimizations can be safely and efficiently applied to real programs, and 6) evaluating the effectiveness of data layout optimizations for real programs.

To evaluate the impact on the performance on scientific applications, I plan to develop and implement data layout optimizations as passes in the Stanford SUIF compiler. A prototype implementation can already modify variable base addresses and perform array transpose automatically. I will test the data layout compiler optimizations on programs from benchmark suites such as Perfect, NAS, SPECfp and SPECint. I will measure the impact of the optimizations on application performance on a variety of scalar and parallel architectures, and use simulations to precisely determine the effect of optimizations. Preliminary experiments on the DEC Alpha are very promising. For *swm* and *tomcatv*, two programs from SPEC, the prototype compiler improved sequential performance by 29% and 12%, respectively.

This project will make several important contributions, including: 1) sophisticated data layout optimization techniques 2) extensive experimental evaluation on scalar and parallel architectures 3) quantitative measurements of the impact of these optimizations 4) insights towards improved computer architecture design The compilation techniques developed by this project will be of great interest to computer vendors. My findings on the impact of data layout will also help computer architects. Most importantly, by making it easier to automatically exploit computation power in high-performance architectures, my compilation techniques will make computational science more accessible for scientists and engineers.

Contents

Project Description

1 Introduction

High-performance computing is rapidly becoming an important component of scientific research and development. Today's scientists and engineers depend on the computing power of supercomputers and fast workstations to solve many important problems in fields such as aeronautics, physics, biology, and medicine. A major factor in the success of scientific computing is the rapid increase in computing capability. Computation power has increased dramatically due to advances in two areas. First, higher chip density and improved processor designs have made individual microprocessors more powerful than the supercomputers available just a decade ago. Second, the emergence of parallel computing has allowed scientists and engineers to harness large numbers of commodity microprocessors together to solve large problems quickly and cost-effectively.

A key contributor to the success of computational science has been the development of software which allows users to run their programs efficiently on high-performance architectures (both sequential and parallel) without excessive concern with the details of the underlying hardware architecture. In particular, compilation technology has advanced significantly, especially in its ability to improve performance by restructuring the computation executed by a program. Compilers are currently sufficiently successful that programmers are freed from explicitly managing hardware features such as multiple functional units, speculative execution, out-of-order instruction issue, vector registers, and fine-grain loop parallelism. Ongoing research on advanced program analyses may soon also allow compilers to also hide details of coarse-grain parallelism and interprocessor communication.

Despite these advances, major trends in computer architecture design demand that additional compiler technology be developed and deployed in commercial systems. In particular, the growing disparity between processor and memory speeds have motivated computer architects to create in modern processors a *memory hierarchy* where higher levels (e.g., registers, primary cache) are fast and small, while lower levels (e.g., secondary cache, memory) are slow and large. The ability of an application to exploit reuse of higher levels of the memory hierarchy thus becomes critical for achieving good performance.

For more concrete examples of how memory latencies affect performance, consider recent microprocessors such as the DEC Alpha or the MIPS R10000. Data in primary (on-chip) cache can be accessed in a few cycles, secondary (off-chip) cache accesses take 10–20 cycles, and local memory accesses take 60–100 cycles. Applications that do not use caches effectively thus spend most of their time waiting for memory accesses. Processors can now issue multiple instructions per cycle, making long memory latencies even worse obstacles to good performance.

Two recent simulation studies of document the importance of memory access times on overall performance of scientific applications. Mowry *et al.* discovered programs approximate spend from a quarter to half of overall execution time waiting for data to be fetched from memory during sequential execution [43], shown in Figure 1. Torrie *et al.* found similar results for compiler-parallelized applications during execution on a 16-processor parallel machine [47], shown in Figure 2. Obviously, not all memory system overhead can be eliminated by the compiler, but these studies point out the importance of reducing memory costs when possible for both scalar and parallel machines.

1.1 Data Locality and Cache Effects

In order to exploit the greater speed of higher levels of the memory hierarchy despite their smaller size, applications must possess *data locality*, where an application reuses data already brought in. Reuse may be in the form of *temporal locality*, where the same data is accessed multiple times, or *spatial locality*, where nearby data is accessed together. With respect to the program, locality may be in the form of *self-reuse* caused by a single variable reference, or *group-reuse* caused by multiple variable references.

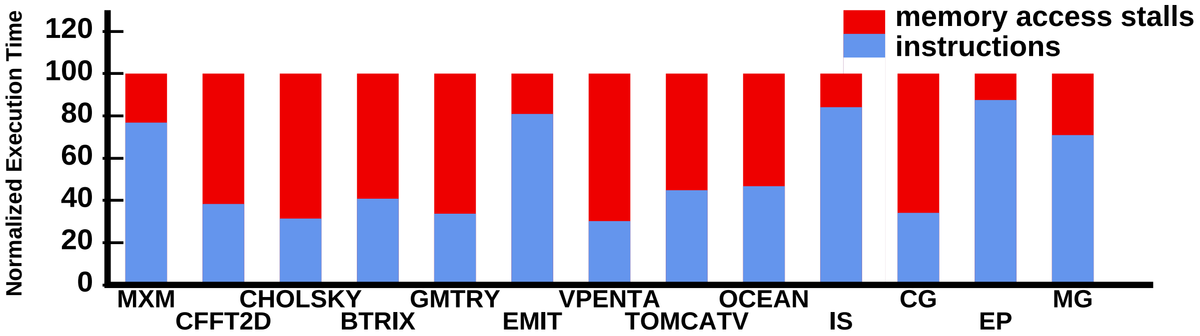

Figure 1 Memory system overhead for 1-processor sequential execution

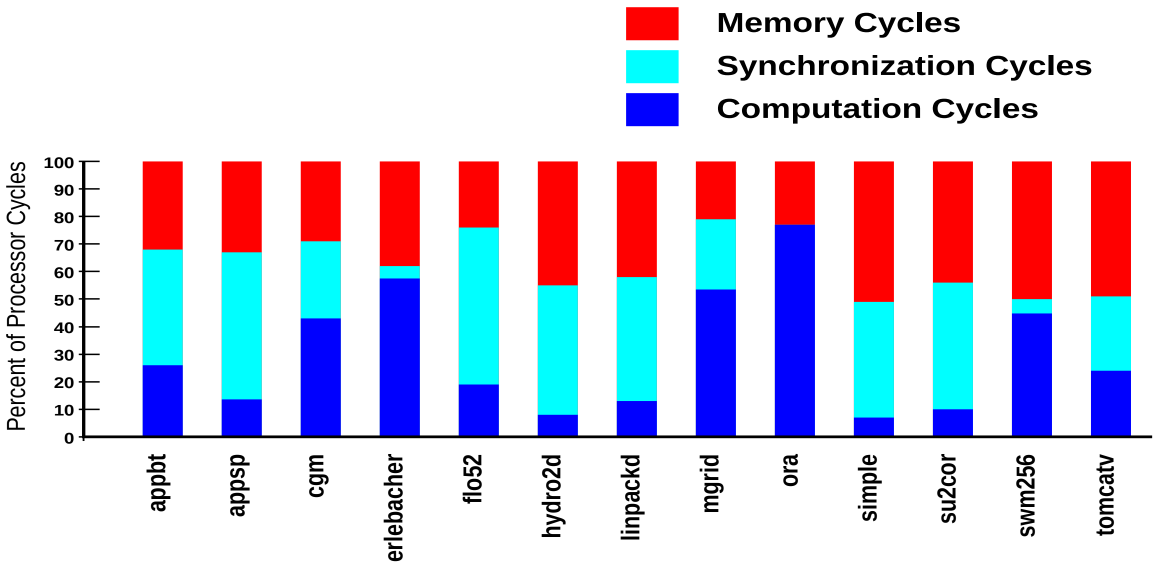

Figure 2 Memory system overhead for 16-processor parallel execution

Often hardware limitations would prevent applications from exploiting reuse, especially of caches. Because of physical considerations, caches are structured a finite collection of multi-unit *cache lines* with a small fixed *set associativity*. This structure may reduce the ability of an application to utilize the cache through a number of problems commonly known as *cache effects*, including the following. *Capacity* misses are caused when the limited size and replacement policy of a cache causes cache lines to be flushed from the cache before they can be reused. *Conflict* misses are caused when the limited set associativity of a cache forces a cache lines to be flushed from cache before it may be reused, despite sufficient capacity.

For parallel applications, *true sharing* misses occur when writes performed by other processors force data to be flushed from cache to maintain coherence. *False sharing* misses occur when writes to logically distinct but physically co-located data on a cache line by another processor force data to be flushed [6, 11, 46]. All types of cache misses must be considered when attempting to improve program performance. In particular, poor spatial locality increases the chance of conflict misses and false sharing.

1.2 Motivating Examples

To see how cache effects may reduce locality exploited by the cache, consider the following simple examples. For instance, in the following loop performing a vector dot product there is temporal locality for the reference to S and spatial locality for the references to B(i) and C(i), leading to cache reuse.

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

However, consider what happens when the cache is direct mapped and the base addresses of B and C are placed apart at a multiple of the cache size. Every access to C(i) will be mapped to the same cache line as the previous access to B(i) and vice versa, causing every reference to be a conflict miss. Cache effects between variables thus eliminate spatial reuse.

In a similar fashion, consider the following Jacobi iteration kernel, a stencil computation used to iteratively solve partial differential equations. There is much temporal and spatial reuse found in the group of references to B.

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

However, If B is a large array, where each column is a multiple of the cache size, then different columns of B will be mapped to the same cache lines, causing conflict misses. Cache effects thus eliminates spatial reuse within a variable. These two examples demonstrate cases of *severe* conflict misses, both between variables and within variables. In such cases, the conflict occurs so quickly after each reference that locality is lost. Cache lines are flushed out of cache before data on the cache line can be reused. Both examples demonstrate loss of spatial locality, but temporal locality is lost as well.

In comparison, the following example shows a case of *mild* conflict misses, where spatial locality is exploited, but some temporal locality is lost. Assume that the cache is precisely large enough to hold $2N$ array elements. The location of X is enough to offset the base addresses of B and C sufficiently so that the entire cache line is used. However, eventually most cache lines for B will be flushed due to accesses to C, causing conflict misses on the next iteration of the outer k loop.

```
real S, T, B(N), X(32), C(N)
do k = ...
  do i = 1,N
    S = S + B(i)*C(i)
```

The total number of cache misses caused by mild conflict is smaller than that caused by severe conflict (by a factor of the cache line size in number of elements). However, it is worth eliminating nontheless.

2 Data Layout Optimizations

Within the past few years, researchers have discovered program transformations that can be used to improve the data locality of applications. Examples include loop permutation, tiling, loop fission, and loop fusion. These *computation-reordering* optimizations are generally based on changing the order iterations of a loop nest are executed, and are guided by a model of likely cache misses for different configurations of each loop nest [8, 55]. However, researchers are beginning to discover that for many applications, loop transformations alone may be not sufficient for achieving good data locality. For instance, for the previous on motivating examples, program transformations that reorder computation would have little or no effect on reducing conflict misses. Instead, a new class of data layout optimizations may also be essential for efficiently using the memory hierarchy.

In comparison to compiler optimizations which reorder the computation performed by the program, *data layout* optimizations modify how static and heap allocated variables in a program are laid out in memory, with the goal of improving spatial locality and avoiding adverse memory effects such as conflict misses and false sharing [3, 9, 28]. Conflict misses are particularly important as cache sizes grow larger. In a recent study of uniprocessor cache behavior of scientific programs McKinley and Temam found most cache misses within loop nests and half of all misses to be caused by cache conflicts [42].

Data layout optimization techniques include modifying the distance between variables, changing the size of array dimensions, transposing array dimensions, partially linearizing array dimensions, rearranging fields of structures, and modifying dynamic memory allocation policies. Most optimizations must be applied at compile time, but link-time and run-time optimizations are also possible. Some optimizations are designed for array-based codes, while others target codes with pointers and complex data structures. While individual data layout optimizations have been proposed and studied to a limited extent, there is yet no comprehensive examination of their efficacy for large programs, or when applied in conjunction with each other and computation-reordering locality optimizations. The goal of this proposal is to develop and evaluate a comprehensive approach for applying data layout optimizations.

Note that a disadvantage of all data layout optimizations is that effects could be uncertain. Transformations to eliminate cache conflicts may actually cause new conflict and capacity misses. As a result sophisticated compiler algorithms and precise analyses are very important. In the following sections, I first describe individual data layout optimizations discovered by previous researchers, then point out the important unexplored issues that need to be resolved. I then present an optimization strategy and a plan for evaluating its effectiveness.

2.1 Variable Base Addresses

The simplest data layout optimization is to simply change the base address of statically allocated variables. The order variables are laid out in memory can be changed, and unused variables (pads) inserted. A simple example is shown in Figure 3. The most important effect is to modify the difference between base addresses of two variables so that they no longer cause cache conflicts. Other uses include co-locating related variables to exploit spatial locality, aligning variables to memory hierarchy units to improve spatial locality and avoid false sharing. Disadvantages include wasted memory for pads.

Base addresses of global variables may be modified by the linker. Addresses of local variables can only be modified by the compiler. Addresses of dynamically allocated heap objects may be selected by the run-time memory management system. It is interesting to note that good heuristics have not been published for even this simple data layout optimization.

2.2 Variable Field Offsets

A related optimization is to analyze and modify the offsets of fields of a structure/record, changing their order and inserting unused fields (pads) where necessary. The effects are the same as for individual variables, but as applied to fields within a variable. A simple example is shown in Figure 4. Advantages are the elimination of conflict misses between fields of a structure, as well as co-location of related fields for improved spatial locality and possibly alignment of fields to memory hierarchy units. Disadvantages include extra memory needed for pads. Modifying variable field offsets can only be performed at compile time.

2.3 Group-and-Transpose/Split-and-Scatter

Group-and-transpose is a method used to improve spatial locality between elements of different arrays or structures [28]. A simple example is shown in Figure 7. If elements from different data structures are distant but usually accessed together, transforming the array or data structure containing the elements can change the data layout so that the elements are now contiguous, improving spatial locality and indirectly reducing conflict misses. Group-and-transpose must be applied at compile time. Disadvantages potentially include increased address calculation costs, particularly for pointer-based data structures.

Note that the reverse of group-and-transpose may also be a useful program transformation for improving spatial locality. If only part of a compound data structure is being consistently accessed, applying *split-and-scatter* can avoid fetching data not needed.

2.4 Array Padding

Array padding differs from modifying variable base addresses in that it increases internal array dimension sizes, changing the relative layout for higher dimensions of the array [5, 45]. A simple example is shown in Figure 6. Unlike changing the base or field address of variables, array padding can eliminate conflict misses between different sections of the same array. Disadvantages include extra memory for pads within the array. Array padding must be performed at compile time. It is interesting to note that in many scientific applications, users have already padded arrays by hand to avoid conflict misses. However, no algorithms have been published for selecting safe array paddings.

2.5 Array Transpose

Array transpose modifies the relative ordering of array dimensions. It changing the spatial locality of array dimensions, and can also indirectly reduce conflict misses [9]. A simple example is shown in Figure 7. Array transpose must be performed at compile time. No additional memory is required, but the significant rearrangement of data layout with the array may cause new cache misses.

2.6 Array Reindexing

Array reindexing combines array transpose with tiling of array dimensions in a manner that allows piece-wise multidimensional rectangular slices of an array to be linearized in the address space [3, 51]. A simple example is shown in Figure 8. It dramatically modifies the layout of an array, and can significantly improve spatial locality for chosen sections of an array, and indirectly reduces capacity and conflict misses. Array reindexing must be performed at compile time. It may require additional memory, and will greatly increase address calculation costs unless special compile-time optimizations can be performed.

2.7 Heap Allocation

Except for modifying variable field offsets, previous data layout optimizations all affect statically allocated data. It is also possible to modify heap memory allocation routines to improve spatial locality or reduce conflict misses. Memory can be allocated in non-conflicting locations, or multiple heap allocations may be located closely together to improve spatial locality. A simple example is shown in Figure 9. Heap allocation is performed at run time. Compile-time analyses and transformations are not required, but are probably needed for improved results.

2.8 Page Coloring

Page coloring is a technique to improve multiprocessor data layout at the page level during run time in the operating system [7]. Assuming the multiprocessor possesses a physically indexed cache significantly larger than the page size, the operating system can choose a page mapping of virtual to physical addresses that reduces cache conflict between pages. Compiler analysis is required to tell the operating system which pages may conflict and should be mapped to different locations (colors).

Unlike the optimizations discussed previously, I do not intend to evaluate the effectiveness of page coloring. Instead, I believe that many of the benefits of page coloring can also be achieved by a combination of modifying variable base addresses and array padding. I propose to compare their effectiveness with page coloring for the applications used in the page coloring study.

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

Figure 3 Variable Base Address

```
common /foo/ A(N),B(N)    -->  common /foo/ A(N),PAD(M),B(N)

struct {                  -->  struct {
   float A[N];                    float B[N];
   float B[N];                    float PAD[M];
};                                float A[N];
                               };
```

Figure 4 Variable Field Offsets

```
real A(100)              <-->  real AB(2,100)
real B(100)

float A[100];            <-->  struct {
float B[100];                     float A;
                                  float B;
                               } AB[100];
```

Figure 5 Group-and-Transpose/Split-and-Scatter

```
real A(M,N)               -->  real A(M+P,N)
```

Figure 6 Array Padding

```
real A(M,N)               -->  real A(N,M)
```

Figure 7 Array Transpose

```
real A(M,N)               -->  real A(M/P,N,P)
```

Figure 8 Array Reindexing

```
// avoid conflicts
float *p = alloc(32);     -->  float *p = alloc(34);
float *q = alloc(32);          float *q = alloc(34);

// improve locality
float *p = alloc(32);     -->  float *p = alloc(64);
float *q = alloc(32);          float *q = p+32;
```

Figure 9 Heap Allocation

3 Preliminary Evaluation

3.1 Prototype Implementation

To establish the potential impact of data layout optimizations, I have implemented simple versions of two optimizations (variable base address, array transpose) as optimization passes in the Stanford SUIF compiler [20, 53]. SUIF takes as input Fortran or C programs, and can generate transformed C, Fortran, or assembly code for workstations. Simple heuristics were used to guide transformations, given the lack of precise compile-time analysis available in the prototype compiler.

Performing array transposes automatically is relatively simple, requiring modification of array declarations and all references to the transposed array. Conservative safety checks can be performed simply and efficiently. Implementing variable base address selection is more complicated. In order to select base addresses of variables at compile time, the SUIF compiler performs two transformations. First, it promotes all local variables into the global scope. This process Actual parameters to functions do cannot and do not need to be promoted, since they represent variables declared elsewhere.

The second compiler transformation is to take all global variables and make them into fields of `suifmem`, a large structured variable (or common block in Fortran). This transformation results in a single global variable containing all of the statically allocated shared memory in the program, as shown in the example below. Now the compiler can modify the base addresses of variables as needed by reordering fields in the `suifmem` structure, inserting pad variables where needed. A similar transformation is made by SUIF in order parallelizing programs for the heavy-weight thread model employed by software DSMs [31, 32].

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

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

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

3.2 Experimental Evaluation

The prototype optimizer was used to transform a number of sequential and parallel kernels and benchmark programs. The original and optimized versions of each program were then executed on DEC Sables multiprocessors with four 275MHz Alpha 20064 processors and 256 megabytes of memory each, operated under DEC Unix version 3.2D. The Alpha processors had 16KB on-chip direct mapped primary caches, with 32B cache lines.

Due to the preliminary state of our prototype implementation, we were unable to collect experimental results for all optimization options. For many test cases, optimizations had little effect on performance. However, dramatic improvements were achieved for certain programs as shown in Table 1. These encouraging results make further investigation of data layout optimizations a high priority.

Program	Size	Description	% Improvement from Optimization			
			Variable Base Address		Array Transpose	
			1 proc	4 proc	1 proc	4 proc
expl	256^2	Explicit Hydrodynamics	29%	18%	-	-
	512^2		33%	-	-	-
jacobi	1024^2	Jacobi Iteration	39%	-	-	-
mult2	512^2	Double Matrix Multiply	0%	-	96%	96%
mgrid	32^5	Multigrid Solver	0%	3%	-	-
rb	1024^2	Red-Black Relaxation	1%	4%	-	-
swm	256^2	Shallow Water Model	29%	0%	-	-
	512^2		25%	-	-	-
tomcatv	512^2	Vectorized Mesh Solver	12%	-	-	-

Table 1 Experimental Timings for DEC Alpha

4 Proposed Research

I have shown that exploiting the memory hierarchy is a very important problem for both scalar and parallel machines. A new class of data layout optimizations have been developed to solve parts of the whole problem, but are not well understood. I propose to comprehensively evaluate the effectiveness of data layout optimizations for applications on both scalar and parallel machines, developing new techniques where needed. I will focus mainly on scientific applications, but will also investigate the applicability of data layout optimizations for integer and pointer-based codes. This research wll answer many questions on the ability of system software to avoid adverse memory effects and has great potential for improving performance of both sequential and parallel applications. The main technical goals of my proposal are:

1. Developing heuristics to guide individual data layout optimizations for both scalar and parallel machines.
2. Determining the precision of compile-time analyses needed to effectively guide these heuristics.
3. Combining all data layout optimizations in a single optimization algorithm.
4. Integrating data layout and computation-reordering optimizations in the same optimization framework.
5. Developing analyses to ensure data layout optimizations can be safely and efficiently applied to real programs.
6. Evaluating the effectiveness of data layout optimizations for real programs when dealing with issues such as procedures, resource conflicts, and imprecise analysis.

4.1 Developing Optimization Heuristics

Heuristics for some data layout optimizations have been presented by previous researchers, but usually in a limited context, focusing on either sequential or parallel codes. I intend to develop heuristics designed to achieve the following objectives, in order of importance: 1) eliminate conflict misses, 2) improve spatial locality, 3) eliminate false sharing misses (multiprocessor only), 4) avoid computation overhead, 5) avoid memory overhead.

These objectives were selected because data layout transformations appear to be generally most useful for eliminating conflict misses and improving spatial locality in a program. From previous experimental studies it appears that conflict misses have the most significant impact on program performance for both sequential and parallel programs. In particular, compiler algorithms need to be developed to detect and eliminate *severe* conflict misses which prevent caches from exploiting spatial or temporal locality. The next goal is to improve spatial locality, which will increase reuse and indirectly reduce conflict and capacity misses for both sequential and parallel programs. Data transformations have been shown to be useful for eliminating false sharing. When applying these optimizations, care should also be taken to avoid unnecessary computation and memory overhead caused as side effects.

Predicting Conflict Misses A common requirement for guiding all data layout optimizations is compile-time analysis capable of estimating conflict misses. Several models have been developed to calculate spatial and temporal locality for sequential programs [8, 12, 17, 55]. Analyses have also been developed to analyze programs and select computation/data partitions that attempt to maximize local data accesses for parallel programs [4, 40, 33]. However, none of these models are suitable for predicting the occurrence of conflict misses.

To predict severe conflict misses, I propose a modification of the algorithm by Gannon *et al.* for calculation of *uniformly generated references* [17]. Two references to the same variable are uniform if their subscripts have the same index variables and differ only by a constant term. For instance, assuming `i, j` are loop index variables, `A(i,j)` and `A(i-1,j+2)` are uniformly generated references, but `A(i,j)` and `A(j,i)` are not. The intuition is that references in the same uniformly generated group have very similar memory access patterns that may significantly overlap. Both Gannon *et al.* and Wolf and Lam use uniformly generated references as way of calculating group-reuse when estimating cache locality [17, 14, 55]. Carr *et al.* utilize a similar concept called *reference groups*, customized for individual loops.

To check for severe conflict misses within an array, the compiler can examine members of a group of uniformly generated references to determine whether they differ in non-contiguous array dimensions. If they do, then conflict misses are likely if the size of intervening array dimensions is a multiple of the cache size. To check for conflict misses between variables, the algorithm needs to be modified to allow different variables to be placed in the same group of uniformly generated references. For instance, `A(i,j)` and `B(i-1,j+2)` should be considered together. Assuming their array dimensions are conforming, their access patterns are sufficiently similar so that severe conflict misses can result if the base addresses of `A` and `B` differ by a multiple of the cache size.

Once potential conflict misses are found, they may be factored into the cache cost models used to evaluate the program. If conflict misses occur, they reduce the locality exhibited by a variable or group of variables. The cache cost model can account for the conflict misses by eliminating all or part of the self and group reuse computed for the references.

Initial Proposed Heuristics Given the large number of individual optimizations, it is not feasible to describe each in detail. Here I give a brief sketch of initial proposed heuristics for each optimization. The heuristics will be implemented, tested, and modified when necessary.

- *Variable base addresses.* Identify potential conflict misses between variables for individual loop nests. Construct conflict graph in the same manner as hierarchical register allocation, weighting each loop nest by estimated cost of conflict misses. Modify base addresses for variables through reordering and insertion of padding variables. Ensure difference in variable base addresses for conflicting variables is not a multiple of the cache size. Also identify variables frequently accessed together and place them nearby to enhance spatial locality. Align arrays and compound data structures with cache lines (and pages) where possible to improve cache line utilization and reduce number of misses. For a pair of variables written to by differing processors in parallel programs, ensure pair of variables are placed on different cache lines (or pages).

- *Variable field offsets.* Perform analysis and optimization similar to that for modifying variable base addresses, but apply only to fields within a structure or common block.

- *Group-and-transpose/Split-and-scatter.* Identify arrays or compound data structures consistently accessed together for each loop nest. Determine whether conflicts exist between loop nests, resolve by weighting loop nests with cost of memory references in loops. If spatial locality can be improved significantly, apply group-and-transpose to bring data closer together in memory. To apply split-and-scatter, look for structures which are consistently only partially accessed, then split off the accessed portion into a separate structure.

- *Array padding.* Identify potential conflicts between sections of the same array for each loop nest. Estimate cost of conflict misses for loop nests, then calculate padding for interior array dimensions sufficient to avoid severe conflict misses for as many loop nests as possible. Select small padding sizes to reduce memory overhead. Attempt to avoid introducing new conflict misses. When used in conjunction with tiling, pad array dimensions as necessary to reduce conflict misses within tiles [10, 38]. For array dimensions traversed by parallel loops, decide whether padding dimension to an integral number of cache lines will reduce false sharing.

- *Array transpose.* For each loop nest, discover array dimensions traversed by innermost loops. Attempt to transpose arrays so that those dimensions are contiguous, improving spatial locality. Determine conflicts between loop nests, resolve by weighting loop nests with cost of memory references in loops. For array dimensions traversed by parallel loops, attempt to transpose arrays so those dimensions are as non-contiguous as possible to reduce potential false sharing.

- *Array reindexing.* Algorithms have been previously described for parallel programs [3], so I will focus on the potential usefulness of applying array reindexing to reduce conflict misses for tiled loop nests. When array padding is not sufficient, reindexing individual tiles may be sufficient to eliminate conflicts within tiles [10, 38].

- *Heap allocation.* Identify heap allocated data which is accessed in patterns which cause conflict misses or exhibit poor spatial locality. Modify the heap allocation policy for the affected data through padding or grouping to improve performance.

4.2 Determine Precision of Compiler Analyses

An important issue is the precision of compiler analyses needed for data layout optimizations to be effective. For some optimizations, simple heuristics based on array size may able to achieve good results. Other optimizations, on the other hand, seem to require very precise analyses of data access patterns in individual loop nests, collated across multiple procedures, to avoid performance degradation and obtain benefits. An important part of this proposal is to empirically establish the marginal utility of more precise compile-time analyses used to guide optimization heuristics.

4.3 Combining Data Layout Optimizations

Determining how data layout optimizations need to be combined is difficult, as individual optimizations may affect each other. For instance, group-and-transpose changes variables so that variable base address modification is no longer possible. Variable field offset modification or array transformations are needed instead. Similarly, array padding may change the size of an array, changing the calculations for variable base address modifications. I intend to evaluate different combinations and optimization orderings to discover what works best in practice for real applications.

4.4 Integrating Computation-Reordering Transformations

The problem of establishing an optimization framework is magnified when one considers the interactions between computation-reordering and data layout optimizations. Obviously the optimizations need to be integrated, since they affect each others heuristics. Loop permutation and array transpose both attempt to improve spatial locality for arrays. Loop tiling and array padding both attempt to reduce conflict misses. Applying optimizations together can result in conflict of goals and priorities.

4.5 Safety for Real Programs

From one perspective, it is easier to determine the safety of data layout optimizations since unlike computation-reordering transformations, they are unaffected by data dependences. As a result determining whether it is safe to apply to a data layout optimization has usually been ignored or only explored for small programs. However, to make layout optimizations widely applicable, it is necessary to develop safety checks in the presence of features found in real programs such as multiple procedures, parameter passing, aliasing, common blocks, equivalencing, array reshaping, and pointers. Potential aliasing and program assumptions about sequence and storage association may force the compiler to be very conservative in applying data layout optimizations. I plan to discover what analysis techniques to establish safety are feasible for real programs, and when user annotations and directives are required.

4.6 Effectiveness for Real Programs

Finally, data layout optimizations are unlike computation-reordering transformations in that their effects are global. As a result, the effects of data optimizations are spread throughout the program, making it more difficult to pick a single layout that maximizes performance for the entire program. Evaluating data layout optimizations on small program kernels will thus overstate the impact of optimizations, since a layout can be chosen to maximize performance for the kernel without worrying about the effect on other computation in the program.

Effectiveness is additionally curtailed because it becomes more difficult to ensure safety, making optimizations inapplicable even where they can improve performance. As a result data layout optimizations need to be evaluated on large programs in order to realistically evaluate their impact on real programs. I plan to evaluate data layout optimizations on several collections of benchmark programs in order to gain insight on their effectiveness for large codes.

5 Research Plan

5.1 Research Objectives

My goal is to definitively determine the effectiveness of data layout optimizations for both scalar and parallel high-performance architectures. To accomplish this end I need to design an integrated optimization strategy and implement it in a robust compiler to evaluate its success.

My research plan is as follows:

1. Design sophisticated heuristics to guide data layout optimizations for both scalar and parallel high-performance architectures. Develop strategies for different levels of compiler analysis.
2. Experimentally evaluate these optimizations for benchmark programs from Perfect, NAS, and SPEC, as well as other representative scientific and integer applications. Test the compiler output on a wide

variety of scalar and parallel architectures, as well as detailed simulations.

3. Evaluate properties of applications that are either conducive or obstacles to successful compiler optimization, including features of real programs that reduce the effectiveness of data layout optimizations.
4. Determine whether advanced data layout optimizations affect application characteristics enough to change requirements for memory system design in future scalar and parallel architectures.

5.2 Prototype Compiler

I will implement my compilation techniques in the Stanford SUIF (Stanford University Intermediate Format) compiler [44, 53]. SUIF is a free infrastructure designed to support collaborative research in optimizing and parallelizing compilers. Independently developed compilation passes work together by using a common intermediate format to represent programs. SUIF is also very robust for a research compiler. It has been tested and validated on a large number of benchmark suites. SUIF incorporates a number of advanced program analyses that allow it to successfully parallelize many programs. It currently detects parallel loops and reductions, applies interprocedural parallelization and privatization [20], calculates data decompositions [4], and applies simple synchronization [49] and memory layout [3] optimizations. SUIF also includes computation-reordering data locality transformations such as loop permutation and tiling [55]. As a result, SUIF is an excellent framework for implementing and evaluating data layout optimizations for both sequential and parallel applications.

5.3 Experimental Evaluation

I plan to target programs from benchmark suites such as Perfect, NAS, SPECfp; these are well-known programs designed to showcase features of scientific computations. I also plan to test programs from SPECint to evaluate the benefits of data layout optimizations for integer and pointer-based codes. I plan to measure the impact of my optimizations on application performance for both scalar and parallel machines. I will test microprocessors available at Maryland, including the DEC Alpha, Pentium Pro, and IBM RS6000. Parallel machines I will evaluate include the IBM SP-2, DEC Alpha multiprocessor, and clusters of DEC workstations.

I also plan on performing extensive simulations to gather precise information on cache effects, using as my simulation infrastructure the simulators from two previous simulation studies in which I participated [8, 47]. Simulations will allow me to vary architectural parameters more widely, and gain insight as to the efficacy of data layout optimizations for future processors. I will also determine whether data layout optimizations can achieve the same effects as page coloring for the applications tested in the page coloring study [7].

6 Related Work

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 [55]. Most researchers exploring compiler optimizations to improve data locality have concentrated on computation-reordering transformations derived from shared-memory parallelizing compilers [56, 57].

Loop permutation and loop tiling are the primary optimization techniques used [8, 10, 16, 27, 55], though loop fission (distribution) and loop fusion have also been found to be helpful [8]. Wolf and Lam use unimodular transformations (a combination of permutation, skewing, and reversal) and tiling with estimates of temporal and spatial reuse to improve data locality. They prune their search space by ignoring loops that do not carry reuse and loops that cannot be permuted due to legality constraints [55, 54].

Li and Pingali use linear transformations (any linear mapping from one loop nest to another loop nest) to optimize for both data locality and parallelism [41]. They do not propose exhaustive search, since the search space becomes infinite, but transform the loop nest based on certain references in the program. They give no details of their heuristic to order loops for locality.

Many researchers have also examined the problem of deriving estimates of cache misses in order to help guide data locality optimizations [15, 14, 12]. Gannon *et al.* introduce the notion of uniformly generated references as a means of discovering group reuse between references to the same array [17]. 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 [42].

Researchers examining memory system effects for parallel machines have mostly focused on program transformations to eliminate false sharing and co-locate data and computation. Torrie *et al.* investigated cache behavior for compile-parallelized programs and found memory costs were responsible for over a third of overall execution time [47]. Kennedy and McKinley integrate loop transformations which target parallelism and the memory hierarchy [34]. The impact of false sharing has been studied extensively [6, 11, 19, 39, 46]. Program transformations to co-locate data and computation can reduce true sharing misses, and is a topic of active interest to many researchers [4, 40, 33, 52].

Few groups have examined data layout optimizations. Jeremiassen and Eggers have performed the most extensive examination of automatically eliminating false sharing in explicitly parallel programs in a compiler [28]. In contrast, I plan to adapt their group-and-transpose techniques for both sequential and compiler-parallelized programs. Cierniak and Li examined combining array transpose and loop transformations to improve locality for parallel programs [9]. They use an algebraic formulation to combine nonsingular loop transformations with array transpose, but do not discuss detailed heuristics for resolving optimization conflicts when when multiple references to the same variable exist.

Amarasinghe *et al.* demonstrated the utility of array reindexing for parallel applications [3]. They found it to be significant in eliminating adverse cache effects, though specialized optimizations were necessary to reduce computation overhead for modified array subscripts. Bugnion *et al.* presented page coloring as a technique to improve cache utilization on multiprocessors [7]. Cache utilization was dramatically improved for two SPEC95 programs. In comparison to previous work on data layout optimizations, I will develop more precise analyses and optimization heuristics, investigate combining a much larger collection of techniques, and apply the optimizations to both scalar and parallel machines.

7 Impact of Proposed Research

Though some of the compilation techniques described have been previously considered for shared-memory machines, this proposal is first to unify all the techniques in one compiler, as well as the first to combine compiler optimizations. I will also be the first to consider many of these optimizations in the context of both scalar and parallel machines. This project will make several important contributions:

- a sophisticated data layout optimizing compiler for uniprocessors,
- a prototype data layout compiler for multiprocessors,
- extensive experimental evaluation of data layout optimizations,
- insights towards improved memory system design for scientific and integer applications.

The compilation techniques developed by this project will be of great interest to computer vendors. My findings on limitations of memory system effects will also help computer architects and operating system writers. Most importantly, by reducing the effort of achieving high performance for scientific programs, my techniques will make computational science more attractive for scientists and engineers.

Bibliography

References

[1] V. Adve, A. Carle, E. Granston, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, J. Mellor-Crummey, S. Warren, and C.-W. Tseng. Requirements for data-parallel programming environments. *IEEE Parallel and Distributed Technology*, 2(3):48–58, Fall 1994.

[2] S. Amarasinghe, J. Anderson, M. Lam, and C.-W. Tseng. An overview of the SUIF compiler for scalable parallel machines. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, San Francisco, CA, February 1995.

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

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

[5] 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.

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

[7] E. Bugnion, J. Anderson, T. Mowry, M. Rosenblum, and M. Lam. Compiler-directed page coloring for multiprocessors. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, Boston, MA, October 1996.

[8] 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.

[9] 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.

[10] S. Coleman and K. McKinley. Tile size selection using cache organization and data layout. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.

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

[12] 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.

[13] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M. Wu. Fortran D language specification. Technical Report TR90-141, Dept. of Computer Science, Rice University, December 1990.

[14] K. Gallivan, W. Jalby, and D. Gannon. On the problem of optimizing data transfers for complex memory systems. In *Proceedings of the Second International Conference on Supercomputing*, St. Malo, France, July 1988.

[15] K. Gallivan, W. Jalby, A. Maloney, and H. Wijshoff. Performance prediction of loop constructs on multiprocessor hierarchical memory systems. In *Proceedings of the 1989 ACM International Conference on Supercomputing*, pages 433–442, Crete, Greece, June 1989.

[16] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformations. In *Proceedings of the First International Conference on Supercomputing*. Springer-Verlag, Athens, Greece, June 1987.

[17] 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.

[18] G. Goff, K. Kennedy, and C.-W. Tseng. Practical dependence testing. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.

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

[20] 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.

[21] M. W. Hall, S. Hiranandani, K. Kennedy, and C.-W. Tseng. Interprocedural compilation of Fortran D for MIMD distributed-memory machines. In *Proceedings of Supercomputing '92*, Minneapolis, MN, November 1992.

[22] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C.-W. Tseng. An overview of the Fortran D programming system. 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.

[23] 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.

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

[25] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Evaluating compiler optimizations for Fortran D. *Journal of Parallel and Distributed Computing*, 21(1):27–45, April 1994.

[26] S. Hiranandani, K. Kennedy, C.-W. Tseng, and S. Warren. The D Editor: A new interactive parallel programming tool. In *Proceedings of Supercomputing '94*, Washington, DC, November 1994.

[27] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the Fifteenth Annual ACM Symposium on the Principles of Programming Languages*, San Diego, CA, January 1988.

[28] 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.

[29] P. Keleher. Sparks: Coherence as an abstract object. Technical Report CS-TR-3544, Dept. of Computer Science, University of Maryland at College Park, October 1995.

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

[31] P. Keleher and C.-W. Tseng. Improving the compiler/software DSM interface: Preliminary experiences. In *Proceedings of the First SUIF Compiler Workshop*, Stanford, CA, January 1996.

[32] 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.

[33] K. Kennedy and U. Kremer. Automatic data layout for High Performance Fortran. In *Proceedings of Supercomputing '95*, San Diego, CA, November 1995.

[34] 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.

[35] K. Kennedy, K. S. McKinley, and C.-W. Tseng. Interactive parallel programming using the ParaScope Editor. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):329–341, July 1991.

[36] K. Kennedy, K. S. McKinley, and C.-W. Tseng. Analysis and transformation in an interactive parallel programming tool. *Concurrency: Practice and Experience*, 5(7):575–602, October 1993.

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

[38] M. Lam, E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and*

Operating Systems (ASPLOS-IV), Santa Clara, CA, April 1991.

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

[40] J. Li and M. Chen. The data alignment phase in compiling programs for distributed-memory machines. *Journal of Parallel and Distributed Computing*, 13(2):213–221, October 1991.

[41] W. Li and K. Pingali. Access normalization: Loop restructuring for NUMA compilers. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, Boston, MA, October 1992.

[42] 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.

[43] T. Mowry, M. 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.

[44] S. Tjiang, M. E. Wolf, M. Lam, K. Pieper, and J. Hennessy. Integrating scalar optimization and parallelization. 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.

[45] 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.

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

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

[48] C.-W. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*. PhD thesis, Dept. of Computer Science, Rice University, January 1993.

[49] 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.

[50] C.-W. Tseng. Efficient machine-independent programming of high-performance multiprocessors. Technical Report CS-TR-3545, Dept. of Computer Science, University of Maryland at College Park, October 1995.

[51] C.-W. Tseng, J. Anderson, S. Amarasinghe, and M. Lam. Unified compilation techniques for shared and distributed address space machines. In *Proceedings of the 1995 ACM International Conference on Supercomputing*, Barcelona, Spain, July 1995.

[52] S. Wholey. Automatic data mapping for distributed-memory parallel computers. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

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

[54] M. E. Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Stanford, CA, August 1992.

[55] 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.

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

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

[58] M. J. Wolfe and C.-W. Tseng. The Power test for data dependence. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):591–601, September 1992.