Compiler Optimizations for Eliminating Cache Conflict Misses

Gabriel Rivera *Chau-Wen Tseng*

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

Abstract

Limited set-associativity in hardware caches can cause *conflict* misses when multiple data items map to the same cache locations. Conflict misses have been found to be a significant source of poor cache performance in scientific programs, particularly within loop nests. We present two compiler transformations to eliminate conflict misses: 1) modifying variable base addresses, 2) padding inner array dimensions. Unlike compiler transformations that restructure the computation performed by the program, these two techniques modify its *data layout*. Using cache simulations of a selection of kernels and benchmark programs, we show these compiler transformations can eliminate conflict misses for applications with regular memory access patterns. Cache miss rates for a 16K, direct-mapped cache are reduced by 35% on average for each program. For some programs, execution times on a DEC Alpha can be improved up to 60%.

1 Introduction

The increasing disparity between processor and memory speeds has motivated in modern microprocessors 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. For 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. Scientific programs tend to be particularly memory-intensive and dependent on the memory hierarchy. A simulation study by Mowry *et al.* discovered scientific programs approximate spend from a quarter to half of overall execution time waiting for data to be fetched from memory during sequential execution [18].

In order to exploit the greater speed of higher levels of the memory hierarchy, applications must possess *data locality*, where an application reuses data. 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. Fortunately, scientific applications tend to have large amounts of reuse [4].

Due to speed constraints, hardware caches tend to have limited *set associativity*, where memory addresses can only be mapped to one of k locations in a k-way associative cache. On-chip primary caches typically are direct-mapped (1-way set associative). Because of the limited set associativity of hardware caches, programs may be unable to exploit reuse actually present in the application.

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 [17]. In this paper, we show that compiler transformations can be very effective in eliminating *conflict misses* for scientific programs with regular access patterns.

1.1 Motivating Examples

We begin by providing some simple examples to motivate the need for eliminating conflict misses. Consider the loop performing a vector dot product in Figure 1. Unit-stride references by `B(i)` and `C(i)` provide spatial locality, leading to cache reuse. Consider what happens if the cache is direct mapped and the base addresses of `B` and `C` are separated by 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. Conflict misses between variables thus eliminate

```
real S, B(N), C(N)      --> real B(N), DUMMY(PAD), C(N)
do i = 1,N
  S = S + B(i)*C(i)
```

Figure 1 Inter-Variable Padding (Change Base Address)

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

Figure 2 Intra-Variable Padding (Change Column Dimension)

spatial reuse. A technique for eliminating the conflict misses in such cases is to change the base address of statically allocated variables such as `B` and `C` so that they no longer cause conflicts. The base addresses may be directly modified in the linker, or through the compile-time transformation of *inter-variable padding*, as shown in Figure 1.

In a similar fashion, consider the Jacobi iteration kernel in Figure 2. It is 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`. 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. One method for eliminating such conflict misses is to change internal layout of the array `B` so that they no longer cause conflicts. This change requires a compile-time transformation such as *intra-variable padding*, as shown in Figure 2. By increasing the size of the inner dimension, columns no longer map onto each other.

These two examples demonstrate cases of *severe* conflict misses, both between variables and within variables. These misses are severe because they occur on each loop iterations. 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 can be lost as well.

2 Data Layout Optimizations

This paper investigates the effectiveness of compiler transformations for eliminating conflict misses in scientific programs. We are optimistic, since compilation technology has already advanced to the point where users can run their programs efficiently without excessive concern with other details of the underlying hardware architecture such as vector registers, multiple functional units, out-of-order issue, and instruction-level parallelism.

Previous research has shown that compilers can automatically restructure programs to improve the data locality of applications [4, 22]. Example optimizations 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. However, for the motivating examples shown, program transformations that reorder computation would have little or no effect on reducing conflict misses.

To eliminate conflict misses, a new class of *data layout* optimizations are needed. Data layout optimizations such as inter and intra-variable padding modify how 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 or false sharing in parallel programs [1, 5, 15]. In this paper, we focus on data layout transformations to eliminate conflict misses for sequential programs.

Most data layout optimizations can be applied at compile time, but link-time and run-time optimizations (for heap-allocated objects) are also possible. As with many compiler transformations, the effect of data layout optimizations may be uncertain. Transformations to eliminate cache conflicts may actually cause new conflict and capacity misses. As a result sophisticated compiler algorithms and precise analyses may be needed. In this paper, we explore the effectiveness two data layout optimizations: inter and intra-variable padding. We first describe properties of each transformation, then describe basic optimization algorithms implemented in a prototype compiler.

2.1 Inter-variable Padding (Modifying Base Addresses)

Inter-variable padding simply changes the base address of statically allocated variables. The order variables are laid out in memory can be changed, and unused variables (pads) inserted. Inter-variable padding may also be applied to fields of large structure/records (such as Fortran common blocks), changing their order and inserting unused fields (pads) where necessary.

The most important effect is to modify the difference between base addresses of two variables so that they no longer cause cache conflicts. Other potential uses include co-locating related variables to exploit spatial locality and aligning variables to cache lines and pages to improve spatial locality. Disadvantages include wasted memory for pads. Base addresses of global variables may be modified by the linker. Compiler intervention is needed to modify addresses of local variables and fields of structures or common blocks.

2.2 Intra-variable Padding (Modifying Array Dimensions)

Inter-variable padding can eliminate conflict misses between different variables or fields of a structure, but does not affect conflict misses caused by different references to a single array. Another transformation, intra-variable (array) padding, is required. Intra-variable 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 [2, 19]. Intra-array padding can thus eliminate conflict misses between different sections of the same array. Because it also changes the size of an array, intra-array padding also changes base addresses of variables and may achieve benefits similar to inter-variable padding. Disadvantages include extra memory for pads within the array and fragmentation of the useful data in 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.

3 Prototype Compiler Implementation

The data layout optimizations were implemented as passes in the Stanford SUIF compiler [13, 20]. SUIF is a compiler infrastructure designed to support research in both optimizing and parallelizing compilers. Independently developed compilation passes work together by using a common intermediate format to represent programs. SUIF has been tested and validated on a large number of programs. It takes as input Fortran or C programs, and can generate transformed C, Fortran, or assembly code output for a variety of processors.

3.1 Globalization Preprocessing

In order to enable setting of variable base addresses at compile time, SUIF performs a number of transformations. First, it promotes local array/structure variables into the global scope. Actual parameters to functions cannot and do not need to be promoted, since they represent variables declared elsewhere. Second, SUIF splits the fields of structures and Fortran common blocks so that individual fields or variables may be optimized. Finally, it takes array/structure global variables and make them into fields of `suifmem`, a large structured variable (or common block in Fortran). Preprocessing thus results in a single global variable containing all of the variables to be optimized, as shown in Figure 3. SUIF can now modify the base addresses of variables as needed by reordering fields in the `suifmem` structure, inserting pad variables if needed.

3.2 Intra-variable Padding

The compiler next applies intra-variable padding. The current implementation only considers padding of the lowest-order array dimension (columns for Fortran arrays). Because interprocedural analysis is not currently performed, dimensions of arrays passed as procedure parameters (whole or part) are not padded (since padding must be consistent across procedures to be safe). Two simple intra-variable padding heuristics were implemented:

All-pad A parameter n is provided. All lowest-order array dimensions are padded by n elements unconditionally, provided safety conditions are satisfied.

```
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 3 Globalization Preprocessing Output

Calc-pad A target cache size and a desired *minimum-distance* (in bytes) are provided as parameters. The heuristic decides the amount to pad a given array as follows. If the size of the lowest-order dimension (say, the column size) is within *minimum-distance* of a multiple of the cache size, or, in the case that the column size is less than the cache size, if twice or three times the column size is within *minimum-distance* of the cache size, then increase the column size as necessary to eliminate this condition.

3.3 Inter-variable Padding

Finally, padding between arrays and other variables is performed. Variables are first partitioned into groups with members of equal size. Groups are sorted by element size, then passed individually to a procedure which assigns each variable in the group a final base address at some location beyond the storage for the previously assigned variable. Two heuristics are implemented for choosing the address for each member of a group. A target cache size is provided as parameter.

Min-pad A parameter *minimum-distance* (number of cache lines) is provided. The heuristic attempts to pad variables in a group so that they remain *minimum-distance* apart in the cache. To do so, the candidate set of base addresses is restricted to multiples of *minimum-distance*. Then, for each variable in the current group, whenever possible the heuristic choses the next available address from this set which does not map to the same cache location as any of the previously assigned base addresses of variables *in the group*. If no such address remains, the next available address from this set is selected irrespective of previous assignments. The resulting assignment of base addresses is such that all variable base addresses are separated by a distance of at least *minimum-distance* on the cache. Multiple variables from a group of n members can occupy a single region only when $n >$ *cache-size / minimum-distance*, which is rare for small enough *minimum-distance*.

Max-pad The second mode of operation for the inter-variable padding pass is similar, except that the next power of two equal to or exceeding *cache-size / n* is used in place of *minimum-distance*, again where n is the number of variables in the group. Provided this value is big enough to avoid address alignment problems, each variable in the group can occupy its own cache region.

4 Experimental Evaluation

To evaluate the effectiveness of our data layout optimizations, we used SUIF to transform a number of scientific kernels and applications written by hand or taken from the SPEC92 and NAS benchmarks suites, as shown in Table 1. For each program tested, multiple optimized versions were generated by varying several parameters to the optimization pass. These parameters specified the target cache size and the combination of heuristics involved in the padding decisions. The original and optimized versions of each program were then both timed and simulated. Timings were made on 275MHz DEC Alpha 20064 processors with 16KB direct-mapped primary caches with 32B cache lines. Application cache behavior was also simulated for a number of cache parameters using the ATOM binary rewriting tool. The

Program	Lines	Description
KERNELS		
adi32	63	2D ADI Integration Fragment (Liv8)
dot256	32	Vector Dot Product (Liv3)
erle64	612	3D Tridiagonal Solver
expl512	59	2D Explicit Hydrodynamics (Liv18)
irr500k	196	Relaxation over Irregular Mesh
jacobi512	52	2D Jacobi with Convergence Test
linpackd	791	Gaussian Elimination with Pivoting
mult300	29	Matrix Multiplication (Liv21)
rb512	52	2D Red-Black Over-Relaxation
shal512	227	Shallow Water Model

Program	Lines	Description
SPEC92 BENCHMARKS		
doduc	5334	Thermohydraulical Modelization
fpppp	2718	2 Electron Integral Derivative
hydro2d	4392	Navier-Stokes
mdljdp2	4316	Molecular Dynamics (double precision)
mdljsp2	3885	Molecular Dynamics (single precision)
ora	373	Ray Tracing
su2cor	2514	Quantum Physics
swm256	487	Vector Shallow Water Model
tomcatv	195	Mesh Generation
NAS BENCHMARKS		
appbt	4457	Block-Tridiagonal PDE Solver
applu	3417	Lower-Upper triangular PDE Solver
appsp	3516	Scalar-Pentadiagonal PDE Solver
buk	305	Bucket Sort
embar	265	Monte Carlo
fftpde	773	2D Fast Fourier Transform

Table 1 Scientific Programs in Study

resulting measurements demonstrated the potential effectiveness of data layout optimizations and which combinations of heuristics can be most worthwhile.

4.1 Cache Miss Rates

We begin by simulating application miss rates for a DEC Alpha primary cache, a 16KB direct-mapped cache with 32 byte cache lines. Figure 4 compares cache miss rates for the original and best optimized versions of each program. These results demonstrate the usefulness of compiler optimizations for eliminating conflict misses.

Table 2 presents a more detailed look at the effect of data layout optimizations on simulated cache miss rates. Miss rates are presented for the original program (orig) and various combinations of optimization parameters. The first four versions resulted from applying only inter-variable padding using various minimum-distances (2,4,8 lines) or a compiler-calculated distance (calc) using the *Maxpad* heuristic. Two versions resulted from applying only intra-array padding, using either a fixed 4-element pad (4 elems) or a compiler-calculated pad (calc). The last version (both) combined inter-array padding with minimum-distance of 4 cache lines and intra-array padding with a 4-element fixed pad. The difference between the best miss rate achieved and the original miss rate are shown in the second-to-last column (diff). The percent decrease is listed in the final column (%). Figures 5 and 6 presents the miss rate for each program graphically.

Our results identify no consistently superior inter-variable padding heuristic. The *Minpad* algorithm generally outperformed *Maxpad* Padding between arrays by 2,4, and 8 cache lines seems to usually yield the same performance for *Minpad*, with minor perturbations. Padding by 2 cache lines performs very poorly for adi32. Padding by 4 cache lines is bad for mdljdp2. Padding by 8 cache lines causes mdljdp2 and appsp to degrade slightly. Large inter-variable pads selected by the *Maxpad* algorithm generally performed the same or slightly worse than the *Minpad*, except for three programs (mult300, mdljsp2, buk).

Out of the 25 programs used, improvements were observed in 17 of the best optimized versions. In 11 of these, the decrease in the miss rate exceeded 50%. Among the 6 without improvement, 2 exhibited no change, 2 underwent increases in the miss rate less than 2%, and the remaining 2 underwent increases less under 5%. Overall, among intra-variable and inter-variable padding, the optimization with the greatest impact was inter-variable padding. Intra-variable padding offered comparatively little benefit except for erle64, where a fixed 4-element pad was necessary to reduce the miss rate from 17.8% to to 5%. Later we see that intra-variable padding becomes more important as the ratio of array size to cache size increases.

Overall, the average cache miss rate for all programs can be reduced from 19% to 9% using the best optimization heuristic, a reduction of 10%. On average, cache miss rates are reduced by 35% for each program. For many programs, the cache miss rate is reduced by 70-80%.

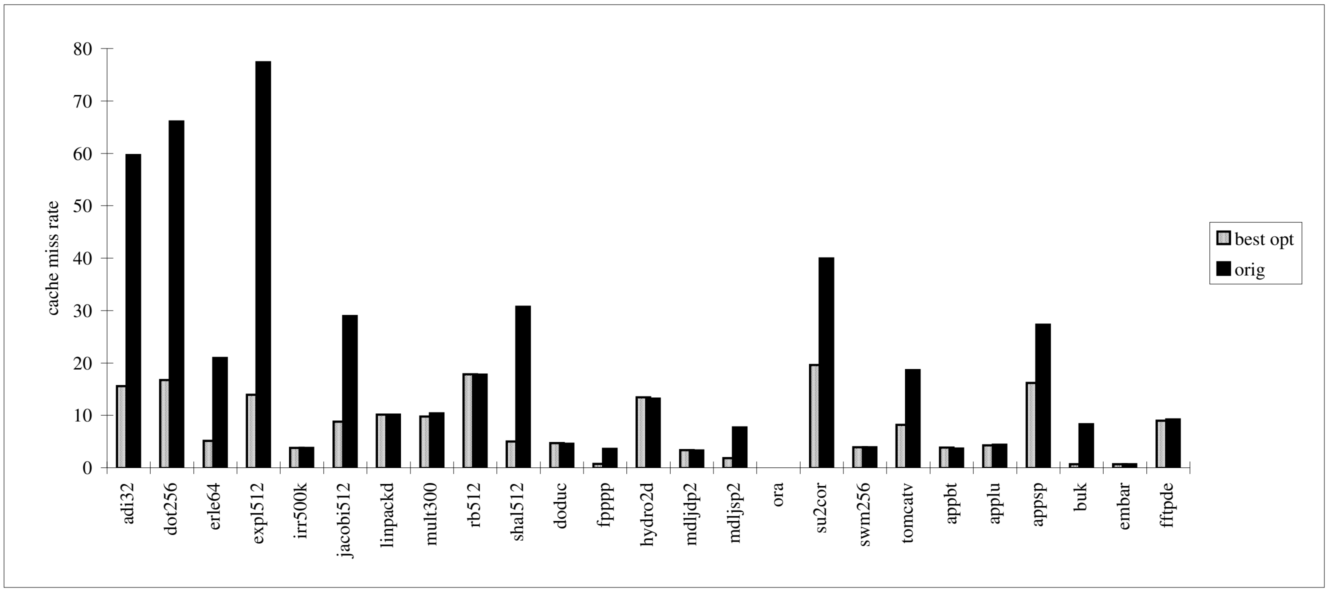

Figure 4 Cache Miss Rate for Original and Best Optimizated Codes

Miss rates (%) for 16K cache, direct-mapped, 32B lines

Program	orig	inter-array				intra-array		both	best improv	
		2 lines	4 lines	8 lines	calc	calc	4 elems		diff	(%)
adi32	59.65	27.98	15.57	15.57	15.57	64.49	59.66	24.76	44.08	73.90
dot256	66.06	16.73	16.73	16.73	16.73	66.06	66.06	16.73	49.33	74.67
erle64	20.96	17.82	17.82	17.83	17.82	17.82	5.14	5.25	15.82	75.48
expl512	77.40	13.90	13.90	13.90	22.79	77.42	72.29	14.01	63.50	82.04
irr500k	3.80	3.80	3.80	3.80	3.80	3.80	3.80	3.80	0.00	0.00
jacobi512	28.93	8.80	8.80	8.80	8.78	28.93	28.94	8.80	20.15	69.65
linpackd	10.14	10.14	10.14	10.14	10.14	10.15	10.15	10.14	0.00	0.00
mult300	10.41	10.40	10.40	10.41	9.77	10.40	10.10	10.10	0.64	6.15
rb512	17.77	17.83	17.83	17.83	17.83	17.83	17.90	17.90	-0.06	-0.34
shal512	30.70	5.03	5.03	5.03	7.15	32.18	32.74	5.06	25.67	83.62
doduc	4.59	5.56	5.56	5.56	5.56	4.80	4.70	5.02	-0.11	-2.40
fpppp	3.59	4.01	4.01	4.01	4.01	0.73	0.80	5.51	2.86	79.67
hydro2d	13.21	13.92	13.92	13.92	14.08	13.42	13.42	13.51	-0.21	-1.59
mdljdp2	3.31	4.51	6.21	5.23	3.57	3.33	3.33	6.20	-0.02	-0.60
mdljsp2	7.69	3.57	3.57	3.57	1.86	2.00	2.00	3.57	5.83	75.81
ora	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
su2cor	39.94	19.62	19.65	19.87	23.08	39.89	39.78	19.68	20.32	50.88
swm256	3.90	3.89	3.89	3.89	6.67	3.90	4.03	4.04	0.01	0.26
tomcatv	18.62	8.17	8.22	8.30	18.71	18.27	9.20	9.33	10.45	56.12
appbt	3.65	4.33	4.33	4.33	4.37	3.86	3.83	4.37	-0.18	-4.93
applu	4.40	4.30	4.30	4.30	4.33	4.63	4.32	4.40	0.10	2.27
appsp	27.30	16.30	16.19	17.05	16.84	32.39	32.42	16.33	11.11	40.70
buk	8.33	1.47	1.47	1.47	0.69	2.11	2.11	1.47	7.64	91.72
embar	0.66	0.67	0.65	0.67	0.67	0.67	0.67	0.67	0.01	1.52
fftpde	9.20	8.97	8.97	8.97	8.97	9.28	9.28	8.98	0.23	2.50
Average	18.97	9.27	8.84	8.85	9.75	18.73	17.47	8.79	10.66	35.71

Table 2 Effect of Optimizations on Cache Miss Rates

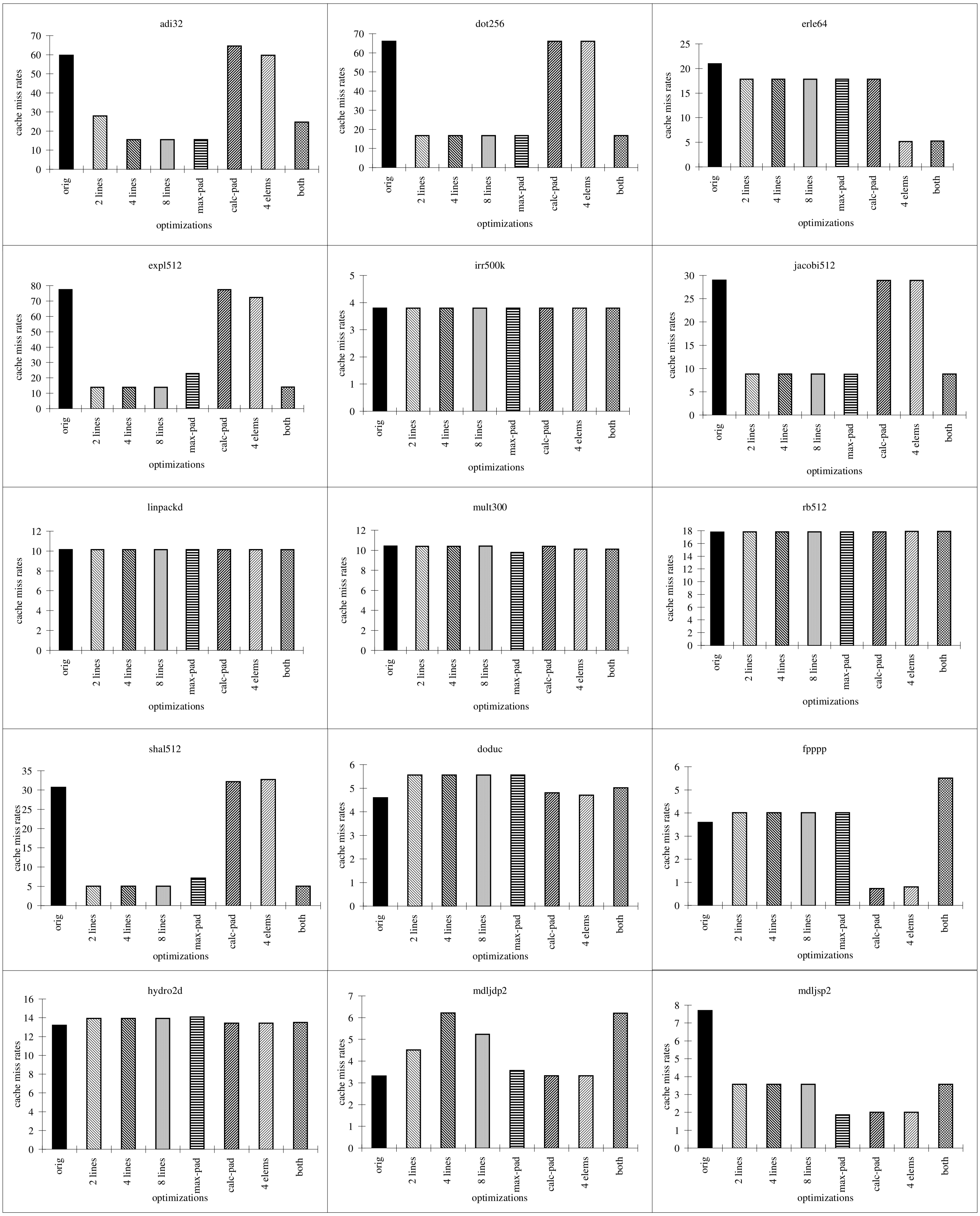

Figure 5 Cache Miss Rate for Optimizations

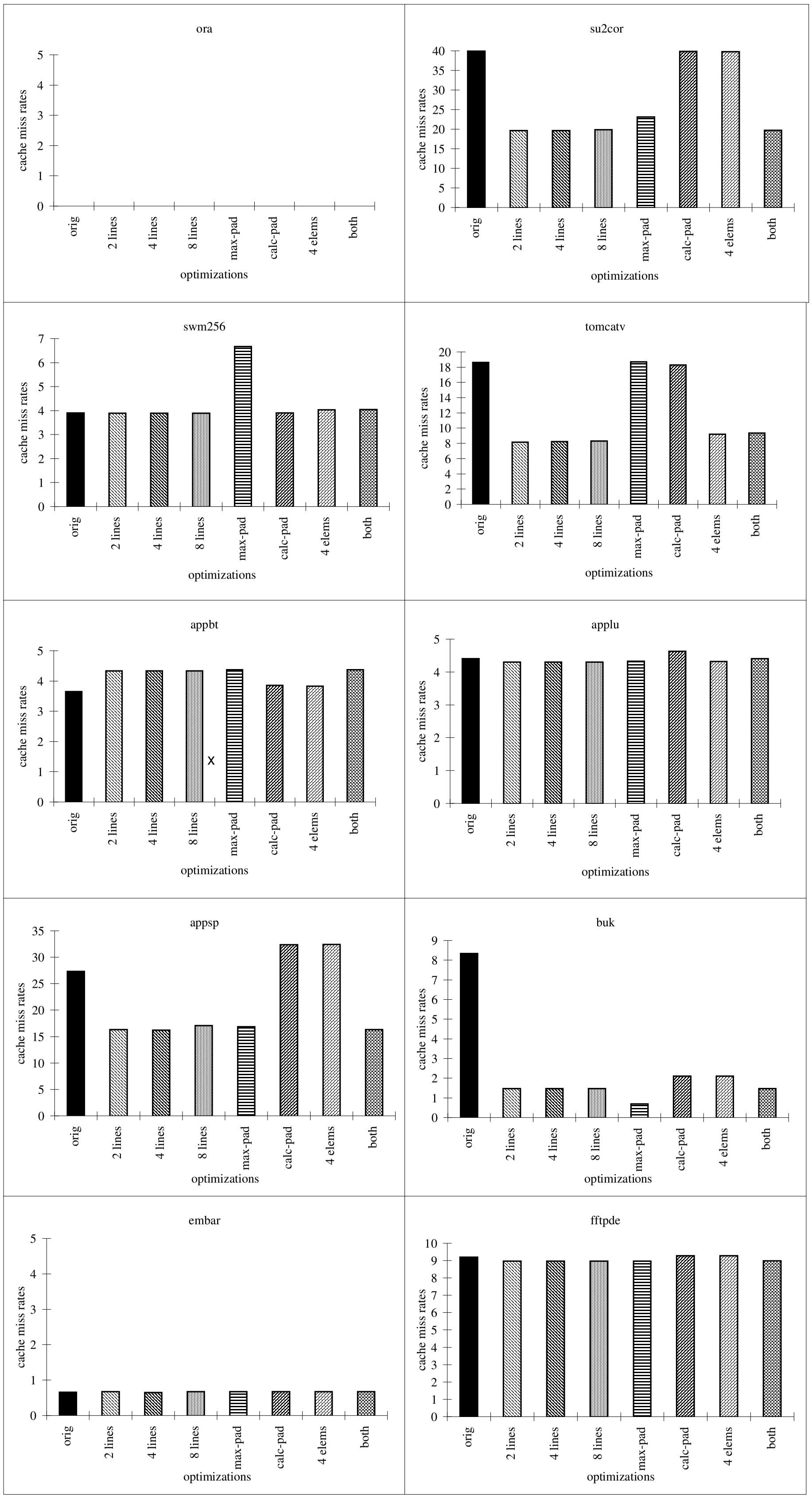

Figure 6 Cache Miss Rate for Optimizations

Miss rates (%) 32B lines

Program	16K direct		16K 4-way		4K direct		4K 4-way	
	orig	opt	orig	opt	orig	opt	orig	opt
adi32	59.65	15.57	14.70	14.95	60.16	18.23	15.37	14.95
dot256	66.06	16.73	16.50	16.50	66.26	17.40	16.50	16.50
erle64	20.96	17.82	4.63	4.63	22.45	20.83	4.64	4.64
expl512	77.40	13.90	60.79	13.73	79.81	47.33	60.96	13.99
irr500k	3.80	3.80	2.23	2.23	25.49	25.50	24.95	24.97
jacobi512	28.93	8.80	8.71	8.71	72.26	52.33	14.49	14.49
linpackd	10.14	10.14	8.33	8.33	15.81	15.81	15.17	15.17
mult300	10.41	10.40	8.39	8.39	14.68	14.61	14.00	13.75
rb512	17.77	17.83	17.77	17.76	70.73	70.69	26.60	26.65
shal512	30.70	5.03	30.19	5.11	60.67	6.33	46.30	23.35
doduc	4.59	5.56	2.87	2.39	15.63	12.38	10.46	7.26
fpppp	3.59	4.01	0.20	0.21	14.87	24.37	9.69	9.54
hydro2d	13.21	13.92	10.71	10.68	16.09	16.19	12.02	12.41
mdljdp2	3.31	6.21	1.90	6.24	12.73	14.97	6.27	9.10
mdljsp2	7.69	3.57	1.09	1.09	14.24	9.58	3.64	2.95
ora	0.00	0.00	0.00	0.00	0.27	0.26	0.00	0.00
su2cor	39.94	19.65	41.18	20.09	66.39	59.73	42.66	23.32
swm256	3.90	3.89	4.06	3.66	45.77	5.94	40.08	4.98
tomcatv	18.62	8.22	8.25	7.99	71.28	23.55	34.86	12.11
appbt	3.65	4.33	2.91	3.08	6.18	6.71	4.11	4.27
applu	4.40	4.30	3.67	3.58	6.77	6.71	3.95	3.86
appsp	27.30	16.19	14.75	10.30	28.29	20.13	16.55	13.59
buk	8.33	1.47	0.50	1.13	2.79	2.89	1.46	3.23
embar	0.66	0.65	0.65	0.65	0.70	0.76	0.65	0.65
fftpde	9.20	8.97	2.30	2.32	9.78	9.91	3.24	3.22
Average	18.97	8.84	10.69	6.95	32.00	20.13	17.14	11.16

Table 3 Effect of Cache Size and Associativity

4.2 Cache Parameters

Table 3 presents the effect of cache size and associativity on simulated cache miss rates. For each cache configuration, we provide the cache miss rate of the original program (orig) and a version with inter-variable padding of 4 cache lines (opt). Results for the 16K cache show that data layout optimizations eliminate conflict misses sufficiently so that for most applications, overall cache miss rates of optimized programs (8.8%) are comparable to those on a 4-way set associative cache (6.9%).

For a few applications, 4-way associativity is not sufficient to eliminate conflict misses. These applications (expl512, shal512, su2cor, appsp) benefit significantly from data layout optimizations even for a 4-way associative cache, heavily influencing the observed overall drop in cache miss rates from 10.7% to 6.9%. The level of miss rates exhibited in these programs are such that miss rates for optimized programs on the direct-mapped cache (8.8%) are actually lower than miss rates for unoptimized programs on the 4-way set associative cache (10.7%).

Comparing 16K with 4K caches, we see while overall cache miss rates increase for 4K caches, data layout optimizations are still important for eliminating conflict misses. Programs such as swm256 which previously have no conflict misses for 16K caches suddenly experience conflicts for 4K caches. However, data layout optimizations are less able to eliminate conflict misses for a 4K cache, as shown by the greater difference between overall cache miss rates of optimized programs (20.1%) and those for a 4-way associative cache (11.2%).

Miss rates (%) for 16K cache, direct-mapped, 32B lines

Program	data set size	orig	inter-array 4 lines	inter-array calc	intra-array 4 elems	intra-array calc	both	best improv diff	best improv (%)
swm	256x256	3.90	3.89	6.67	4.03	3.90	4.04	0.01	0.26
	512x512	14.83	4.09	7.05	4.29	14.83	4.28	10.74	72.42
	1024x1024	44.89	4.99	7.58	5.22	44.89	5.22	39.90	88.88
tomcatv	256x256	18.62	8.22	18.71	9.20	18.27	9.33	10.40	55.85
	512x512	31.90	10.88	19.36	12.70	31.34	10.90	21.02	65.89
	1024x1024	81.96	22.45	28.67	15.35	30.86	11.81	70.15	85.59

Table 4 Effect of Data Set Size

DEC Alpha with 16K cache, direct-mapped, 32B lines

Program	data set	Time (seconds) original	Time (seconds) optimized	improve (%)
adi32		8.30	6.70	19.28
dot256		10.10	8.70	13.86
expl512		32.30	19.50	39.63
jacobi512		11.90	4.50	62.18
mult300		50.00	46.50	7.00
shal512		43.40	31.80	26.73
fpppp		58.70	54.00	8.01
su2cor		165.30	123.80	25.11
swm	256x256	25.30	22.80	9.88
swm	512x512	36.10	30.90	14.40
tomcatv	256x256	33.00	29.00	12.12
tomcatv	512x512	61.50	36.50	40.65

Table 5 Effect of Optimizations on Execution Times

4.3 Data Set Size

Table 4 presents the effect of application data set size on the impact of data layout optimizations. Swm256 and tomcatv, two applications from SPEC92, and their data sets increased by a factor of four and sixteen. We see that as data set sizes increased, more conflict misses result. The impact of data layout optimizations is thus increased. This effect is similar to our previous observation that conflict misses increase as cache size decreases.

For swm256, inter-array padding was not needed for the original 256x256 data set, but reduces miss rates for the larger 512x512 and 1024x1024 data sets. All instances of tomcatv benefit from inter-array padding, but intra-array padding becomes necessary only for the 1024x1024 data set. These results indicate that data layout optimizations may be more important for real applications, which have larger data sets than benchmark programs.

4.4 Execution Times

Table 4 presents the effectiveness of data layout optimizations on program performance. We compared program execution times of the original program and best optimized version on a DEC Alpha. Programs not included did not change in performance. We see that small improvements in primary cache miss rates as measured by the simulator generally did not affect execution times (probably due to the secondary cache). However, programs with large data sets are significantly impacted by reduction in conflict misses. This effect is apparent for the versions of swm and tomcatv with different data set sizes. In fact, timings for the largest versions of swm and tomcatv were not provided because the unoptimized versions took too long to execute!

```
    real*8 za(512, 512), zb(512, 512) ... zz(512, 512)
    do 30 k = 2, N - 1
      do 30 j = 2, N - 1
        zu(j,k) = zu(j,k) + s * (za(j,k) * (zz(j,k) - zz(j+1,k))
          - za(j-1,k) * (zz(j,k) - zz(j-1,k)) - zb(j,k) * (zz(j,k)
          - zz(j,k-1)) + zb(j,k+1) * (zz(j,k) - zz(j,k+1)))
        zv(j, k) = zv(j,k) + s * (za(j,k) * (zr(j,k) - zr(j+1,k))
          - za(j-1,k) * (zr(j,k) - zr(j-1,k)) - zb(j,k+1) * (zr(j,k)
          - zr(j,k-1)) + zb(j,k+1) * (zr(j,k) - zr(j,k+1)))
 30 continue
```

Figure 7 Key Loop Nest in Expl512

4.5 Discussion

From these results, we see that even the simple heuristics implemented in the prototype compiler were able to eliminate cases of severe conflict misses. In general, inserting small inter-variable pads using the *Minpad* heuristic is sufficient to eliminate the worst of the severe conflict misses. Intra-variable padding is rarely needed for the benchmark programs examined, but becomes more important as the ratio of array size to cache size increases. Programs with regular access patterns benefit the most from data layout optimizations. The five programs with irregular data access patterns (irr500k,mdljdp2,mdljsp2,buk,fftpde) either did not benefit or experienced random effects. Programs with few array accesses (ora,embar) were generally unaffected. Only a few programs were greatly improved, but the impact is significant enough that data layout optimizations should be applied.

4.6 Case Study

Our experimental results reveal both the promise and pitfalls of data layout optimizations. Overall, significant improvements are achieved through simple heuristics, but the best combinations of heuristics vary from program to program. A more detailed look at one case illustrates why simple heuristics usually succeed but sometimes fail, and suggests the likely benefits of more sophisticated analyses and more informed heuristics.

The kernels tested in our experiments exhibit many of the characteristics of the larger benchmarks used, but are simpler to understand and summarize. Consider expl512, an explicit hydrodynamics stencil kernel from the Livermore Loops. Figure 7 displays the key loop nest in expl512. Each array referenced inside the loop has dimensions 512 by 512. Clearly, if the arrays are laid out adjacently in memory as declared below, all array base addresses will coincide on any typical cache of size less than 512KB.

One consequence is that references pairs of the form `zz(j,k)`, `zb(j,k)` will collide inside the loop body, resulting in conflict misses. Because cache lines contain multiple array elements, collisions may result whenever references are made to memory locations less than two cache lines apart, modulo the cache size. Hence, references such as `za(j-1,k)` will potentially also cause misses for subsequent accesses to `zz(j,k)`.

These cache effects provide the motivation for inter-array padding. Our experiments demonstrate that pads of 2, 4, or 8 cache lines succeed in reducing the miss rate from 77.4% to 13.9%. However, one of our inter-variable padding heuristics, *Maxpad*, proves considerably less effective, reducing miss rates only to 22.8%. Examining the output code for this heuristic offers insight into why more precise analysis is required.

During optimization, the *Maxpad* algorithm is applied to a single group containing all nine arrays (`za`, `zb`, ... `zz`). Since 16 is the smallest power of 2 greater than or equal to 9, *Maxpad* spaces the arrays $16384/16 = 1024$ bytes apart on the cache. One new and significant unintended consequence is the alignment between *columns* of different arrays on the cache. With the *Minpad* heuristic, the sum of all array pads inserted totaled at most 9 times the size of 8 cache lines, or 9*8*32 = 2304 bytes. Since the column of each array is 4096 bytes in size, references pairs of the form `A(j,k)`,`B(j,k+1)` could never collide, but under the *Maxpad* heuristic, several such pairs exist.

Because conflicts between different array columns, the resulting miss rate is thus higher than when applying *Minpad* heuristic. However, *Minpad* would in fact produce the same undesirable effects given a small enough column size. The common weakness is that neither current inter-variable padding heuristics analyze loop nests. Instead, they assume loop nests access different arrays consistently at similar offsets. Though this assumption holds to a large extent for many scientific programs, it will not always be valid. Our techniques will thus benefit from improved compile-time analysis and more sophisticated heuristics.

5 Improving Compiler Heuristics

The heuristics currently implemented in the prototype compiler are simple and do not perform much compiler analysis. Experimental results indicate that they eliminate the worst cases of severe conflict misses, but are somewhat unstable, causing small perturbations in cache miss rates. In some cases transformations can significantly reduce program performance. More precise compiler analysis can be used to avoid performance degradation, and possibly eliminate additional conflict misses.

5.1 Predicting Conflict Misses

A key requirement for better application of data layout optimizations is compile-time estimation of conflict misses. Several models have been developed to calculate spatial and temporal locality for sequential programs [4, 7, 11, 22]. However, these models are suitable for predicting the occurrence of conflict misses. More recently, Ghosh *et al.* have developed a method for detecting conflict misses by calculating cache miss equations which summarize a loop's memory behavior [12]. They demonstrate how cache miss equations may be used to directly compute the number of conflict misses in a loop nest. We plan to adopt a simplified version of cache miss equations to help guide our array padding heuristics.

To predict severe conflict misses, we propose a modification of the algorithm by Gannon *et al.* for calculation of *uniformly generated references* [11]. 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 [11, 8, 22]. 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.

5.2 Improved Compiler Analysis

Once the compiler has a model for detecting conflict misses, we can use it to determine whether there are sufficient severe conflict misses present to require data layout optimizations. Current heuristics examine the existing global data layout and array declarations to decide whether padding should be performed. More precise analysis may be achieved by also examining the computation and memory access patterns present in the program.

We propose that the compiler identify potential conflict misses between variables for individual loop nests. It can then construct conflict graph in the same manner as hierarchical register allocation, weighting each loop nest by estimated cost of conflict misses. Only arrays that actually conflict with each other are put into the same group to be padded. Once sources of conflicts are identified, we can use current heuristics to select a small set of possible paddings, and use our cache cost model to select between them. Our cost model should prevent significant accidental increases in conflict misses, and by avoiding unnecessary changes to the data layout, we should be able to reduce low-level performance perturbations.

We are currently in the process of incorporating a limited version of *cache miss equations* to detect severe conflict misses and select array paddings [12]. Preliminary results indicate it can consistently match the best result from the set of current compiler padding heuristics.

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 [22]. Gannon *et al.* introduce the notion of uniformly generated references as a means of discovering group reuse between references to the same array [11]. 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 [17].

Most researchers exploring compiler optimizations to improve data locality have concentrated on computation-reordering transformations derived from shared-memory parallelizing compilers [23, 24]. Loop permutation and loop tiling are the primary optimization techniques used [4, 6, 10, 14, 22], though loop fission (distribution) and loop fusion have also been found to be helpful [4]. 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 [22, 21]. 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 [16]. 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 [3, 9, 8, 7]. 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 [12]. They demonstrate how to use cache miss equations to select array paddings to eliminate conflict misses, and block sizes for tiling. We plan to adopt simplified versions of cache miss equations to help guide our array padding heuristics.

Researchers have previously examined changing data layout in parallel applications to eliminate false sharing and co-locate data and computation, have not studied its effect on sequential applications. Jeremiassen and Eggers have performed the most extensive examination of automatically eliminating false sharing in explicitly parallel programs in a compiler [15]. Cierniak and Li examined combining array transpose and loop transformations to improve locality for parallel programs [5]. 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 [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.

7 Conclusions

Conflict misses have been pointed out as a significant source of poor cache performance in scientific programs, particularly within loop nests. In this paper, we present two compiler *data layout* optimizations to eliminate conflict misses: 1) modifying variable base addresses, 2) padding inner array dimensions. Optimizations heuristics currently implemented in the SUIF compiler are naive, but can be improved. Using cache simulations of a selection of kernels and benchmark programs, we show these compiler transformations can eliminate conflict misses for many applications with regular memory access patterns. For some programs, execution times on a DEC Alpha can be improved up to 60%. The compilation techniques and results presented in this paper should be of interest to computer vendors and computer architects. Most importantly, by reducing the effort of achieving high performance for scientific programs, these techniques will make computational science more attractive for scientists and engineers.

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] D. Callahan and A. Porterfield. Data cache performance of supercomputer applications. In *Proceedings of Supercomputing '90*, New York, NY, November 1990.

[4] 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.

[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] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

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

[22] 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.

[23] 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.

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