# Eliminating Conflict Misses for High Performance Architectures

Gabriel Rivera, Chau-Wen Tseng

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

{rivera,tseng}@cs.umd.edu

**Abstract**

Many cache misses in scientific programs are due to conflicts caused by limited set associativity. Two data-layout transformations, inter- and intra-variable padding, can eliminate many conflict misses at compile time. We present GroupPad, an inter-variable padding heuristic to preserve group reuse in stencil computations frequently found in scientific computations. We show padding can also improve performance in parallel programs. Our optimizations have been implemented and tested on a collection of kernels and programs for different cache and data sizes. Preliminary results demonstrate GroupPad is able to consistently preserve group reuse among the programs evaluated, though execution time improvements are small for actual problem and cache sizes tested. Padding improves performance of parallel versions of programs approximately the same magnitude as sequential versions of the same program.

## 1 Introduction

Effectively exploiting caches is widely regarded as the key to achieving good performance on modern microprocessors. Due to hardware constraints, caches have limited *set associativity*, where memory addresses can only be mapped to one of $k$ locations in a $k$-way associative cache. *Conflict misses* may occur when too many data items map to the same set of cache locations, causing cache lines to be flushed from cache before they may be reused, despite sufficient capacity in the overall cache. Conflict misses have been found to be a significant source of poor performance in scientific programs, particularly within loop nests [18]. We previously presented inter- and intra-variable padding, two compiler transformations to eliminate *severe* conflicts, misses which occur on every loop iteration [20]. Unlike standard compiler transformations which restructure the computation performed by the program, these two techniques modify the program's *data layout.*

To see how padding eliminates conflict misses, consider the program in Figure 1. The unit-stride references to A and B provide spatial locality, leading to cache reuse. However, if A and B are separated by a multiple of the cache size in a direct-mapped cache, references A(j,i) and B(j,i) will

```
real A(N,N), B(N,N), C(N,N), D(N,N)

do j = 2,N-1
  do i = 2,N-1
     = A(i,j)
     = B(i,j) + B(i,j+1)
     = C(i,j) + C(i,j+1)
     = D(i,j) + D(i,j+1)

do j = 2,N-1
  do i = 2,N-1
     = A(i,j)
     = B(i,j-1) + B(i,j) + B(i,j+1)
     = C(i,j)
```

Figure 1: Example Program

map to the same cache line in both loops, eliminating reuse. To avoid these severe conflicts, we can apply inter-variable padding to change the base address of B relative to A. Further inter-variable padding can eliminate conflicts between the remaining variables. Similarly, if the column size of B is a multiple of the cache size, columns of B will conflict in both loops, eliminating spatial reuse. To avoid these severe conflicts, we can apply intra-variable padding to the columns of B, changing its internal layout so that columns no longer map to conflicting locations on the cache. There is also *group reuse* of columns of B carried on the outer j loop, since if B(i,j+1) can be kept in the cache it can be reused by B(i,j) on the next iteration of the j loop. Later we present inter-variable padding heuristics to preserve such group reuse.

We implemented inter- and intra-array padding in the SUIF compiler [22]. Preliminary experiments on a collection of kernels and benchmark programs showed padding can lower cache miss rates significantly for some programs [20]. In this paper, we make two new contributions:

- improving padding heuristics to preserve group reuse across outer loop iterations
- evaluating the impact of padding for parallel programs

We begin by presenting previous algorithms for eliminating severe conflict misses, then discuss enhancements to preserve group reuse. We provide experimental results and conclude with related work.

This research was supported in part by NSF grant #CCR9711514 and NSF CAREER Award #ASC9625531 in New Technologies.

In Proceedings of the 1998 ACM International Conference on Supercomputing (ICS'98), Melbourne, Australia, July 1998.

## 2 Padding Heuristics

We present several algorithms for reducing conflict misses. PAD inserts small pads to avoid severe conflict misses. MAXPAD and MAXPADLITE use much larger pads which can improve group reuse for some programs. Finally, GROUPPAD preserves group reuse directly by guiding padding with more precise and powerful analysis.

In this paper we refer to the cache size as $C_s$, the cache line size as $L_s$, and the column size of an array as $Col_s$. The *conflict distance* between two locations is the difference in addresses mod $C_s$. When the conflict distance is less than $L_s$, a conflict miss may arise unless the addresses are actually located on the same cache line. We refer to arrays which have equal element sizes and equal dimension sizes in all but their highest dimension as *conforming*. We also use *uniformly-generated* references [7] in this paper to refer to array references to conforming arrays where array subscripts match excluding constants. For instance, all array references in Figure 1 are uniformly generated. When we linearize uniformly-generated references to compute their conflict distance, loop index variable terms cancel. These conflict distances are therefore constant for a fixed layout, allowing severe conflicts to occur.

We begin by describing intra-variable padding since it applies to all padding algorithms. We then explain each algorithm with respect to inter-variable padding.

### 2.1 Intra-variable Padding

All padding algorithms begin with a common intra-variable padding heuristic, since increasing array dimensions changes array size, affecting variable base addresses. Our heuristic considers sets of conforming arrays simultaneously, padding dimensions of arrays in each identically so that all conforming arrays remain conforming. When two uniformly-generated references to an array in a set have conflict distance $< L_s$, array dimensions are padded to prevent severe conflicts. Increasingly larger pad sizes are used until no further severe conflicts are detected. We also apply previous techniques to pad array column sizes which can lead to frequent conflict misses in linear algebra codes [20].

### 2.2 Algorithm PAD

PAD performs inter-variable padding by assigning base addresses to variables, padding a variable $A$ whenever conflict distances between uniformly-generated references to $A$ and some previously assigned variable are less than $L_s$ [20]. Increasingly larger pad sizes are used until conflict distances are $\geq L_s$. Since pads larger than $C_s$ are equivalent to smaller pads, the heuristic attempts only pads $< C_s$. For all programs tested, PAD requires only a few cache lines of padding per variable.

Figure 2 illustrates the layout achieved by PAD when applied to the sample program of Figure 1, when $C_s$ is about $2\frac{1}{2}$ times the common column size. In our layout diagrams, each box corresponds to the cache during a given loop nest. The width of each box is thus $C_s$, the cache size. Each dot represents a variable reference; its position in a box indicates its cache location inside the loop nest. The vertical line above `A` shows the relative position of `A(i,j)`; other vertical lines are interpreted similarly. As a consequence, vertical lines also reveal relative positions of base addresses. Arcs connect references to the same variable. For instance, the reference connected to `B(i,j)` in the first loop is `B(i,j+1)`. The two

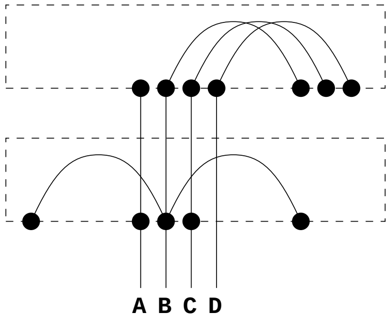


Figure 2: PAD Layout for Example

dots are thus $Col_s = \texttt{N}$ apart. Since all references are uniformly generated, these relative positions do not change over loop iterations.

Layout diagrams are convenient for showing how inter-variable padding can avoid severe conflicts and preserve group temporal reuse. Severe conflicts occur when two references are mapped to the same cache line, and can be prevented by keeping dots at least $L_s$ apart. Group reuse between two columns of an array can be exploited only if the cache lines for the first column are not flushed before they are reused. In our layout diagram, group reuse is represented by having no dots appear between an arc connecting two array columns. To see why, consider that all references move in unit stride between loop iterations. It follows that if a reference is connected by an arc from the right, it reuses the data accessed by its right neighbor only if there are no intervening references "underneath" this arc.

Supposing array sizes are multiples of $C_s$, we find all base addresses in the original sample program coincide on the cache, causing severe conflicts between references to different arrays. An illustration for an unpadded program would therefore need to superimpose references upon on another. No such effects occur in Figure 2. We see PAD eliminates severe conflicts by inserting small pads between successive variables. However, since dots appear between the endpoints of all arcs, no group reuse is exploited. For instance, the reuse of `B(i,j+1)` by `B(i,j)` in the first loop is prevented by `C(i,j)`.

### 2.3 Algorithms MAXPADLITE and MAXPAD

For each variable laid out, PAD uses a small amount of padding to avoid severe conflicts. However, separating variables further in cache may also preserve group temporal reuse between references. Manjikian and Abdelrahman describe an algorithm called *cache partitioning*, in which they space out variables with uniformly-generated references as far as possible in a cache. We implemented MAXPADLITE, a version of their algorithm which maximally spaces sets of conforming arrays. We also enhance the algorithm by combining MAXPADLITE with PAD. The resulting algorithm, MAXPAD, initially separates variables maximally on the cache and then introduces small additional padding to eliminate any remaining severe conflicts.

Figure 3 illustrates the layout achieved by MAXPAD (or MAXPADLITE) when applied to the sample program with a smaller column size. Given a column size somewhat smaller than $\frac{1}{4}C_s$, we see that spacing all base addresses maximally eliminates severe conflicts. Moreover, MAXPAD preserves group reuse which PAD fails to regain in Figure 2. MAXPAD becomes less effective as the column size $Col_s$ approaches

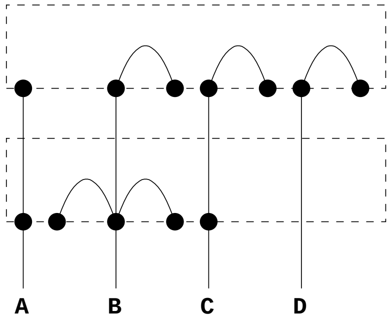


Figure 3: MaxPad Layout for Example

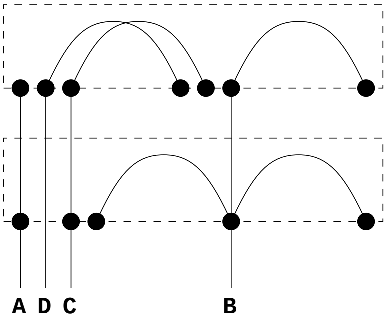


Figure 4: GroupPad Layout for Example

$C_s/V$, where $V$ is the number of variables being padded. As we increase $Col_s$ for the sample program we find that references connected by arcs spread out until severe conflicts result. In Figure 3 for instance, reference `C(j,i+1)` in the first loop will move outward from `C(j,i)` until it overlaps `D(j,i)` for $Col_s = C_s/4$. MaxPadLite, which does not check for severe conflicts, will allow such a layout, whereas MaxPad will insert additional pads to eliminate severe conflicts. However, beyond $Col_s = C_s/4$ neither MaxPadLite nor MaxPad are able to preserve group reuse.

### 2.4 Algorithm GroupPad

Figure 4 demonstrates that there exist layouts for the sample program preserving group reuse where $Col_s$ is larger than $C_s/4$, indicating an algorithm more powerful than MaxPad could be useful. This section presents GroupPad, which uses more sophisticated analysis than MaxPad to achieve the layout of Figure 4. GroupPad lays out sets of conforming arrays in a manner which simultaneously eliminates severe conflicts and preserves group use. Unlike the other algorithms, GroupPad does not set final variable base addresses at each iteration. Instead, it constructs a list of variables whose base addresses can change with each insertion.

GroupPad analyzes a more restricted class of arrays accesses than Pad: references of the form `A(i +` $c$`, j +` $d$ `)`, where $c$ and $d$ are integers with small absolute value. In this section, we refer to $c^r$ as twice the $c$ value found with greatest absolute value, an upper bound on the range of $c$. For simplicity we also limit discussion to the case where $d$ is between -1 and 1. It should then become apparent how to allow for a broader range of values for $d$. While GroupPad fails to analyze all forms of group reuse, it is sufficiently powerful to recognize group reuse in stencil computations. For the rest of this section, the notions of *location* or *base address* will relate to position *on the cache*, not in main memory.

The key GroupPad strategy is to consider only base addresses which are multiples of $Col_s$ and the value *StdPad*. *StdPad* is computed as $L_s + c^r$. Intuitively, *StdPad* values are used to avoid severe conflict misses and $Col_s$ values are used to attain group reuse. In Figure 4 for instance, we can express base addresses in the layout as follows:

$$\begin{array}{cc} A = 0 & D = StdPad \\ C = 2StdPad & B = 3StdPad + Col_s \end{array} \qquad (1)$$

GroupPad obtains such a layout by building a list of *ColPad* symbols and variable symbols. The list begins with a fixed number of *ColPad* symbols determined by $C_s$, $Col_s$, *StdPad* and $V$, the number of variables being padded. Variables are inserted one at a time at a position in the list deemed optimal by GroupPad. The layout for the variables are then computed by traversing the list, computing the address for each symbol as the address plus the *padsize* of the previous variable. The *padsize* of a variable symbol is *StdPad*. The *padsize* of a *ColPad* symbol is $C_s$. The first element has address 0. For example, the following list can derive the base addresses of the layout given in (1):

$$(\mathtt{A}, \mathtt{D}, \mathtt{C}, ColPad, \mathtt{B}, ColPad) \qquad (2)$$

To determine the optimal position at which to place successive variables, all positions are considered and the number of preserved group reuses over all loop nests is computed for each position. The final position is the one maximizing group reuse over all loops. For instance, supposing $Col_s$ is in the range suggested by Figure 4, GroupPad begins with the list (*ColPad*, *ColPad*). Variables `A`, `B`, `C`, and then `D` are inserted at list positions implied by (2). `A` defaults to the first position. Next, `B` is placed after the first *ColPad* to avoid interference from `A`. `C` is optimally placed after `A`. Finally, `D` will eliminate group reuse no matter it is inserted, and so it is placed after `A` where it does the least harm. Three instances of group reuse remain.

We compute $NumColPads$, the number of initial *ColPad* symbols as $\lfloor (C_s - V \times StdPad)/Col_s \rfloor$, intuitively the number of *ColPad*s which can fit on a cache already occupied by $V$ variables. Avoiding severe conflicts in the final layout depends on *StdPad* equaling $L_s + c^r$. In addition, this requires that in the circular list formed by linking the head and tail, a sequence of list elements containing more than $Col_s/StdPad - 1$ variables must also contain two *ColPad* symbols. This restriction can be enforced during variable insertion. Finally, in order for the total *padsize* across the list to equal the $C_s$, additional padding totaling $P$ units is inserted between variables in the final layout so that $VStdPad + NumColPadsCol_s + P = C_s$.

## 3 Impact on Parallel Codes

So far, we have only considered conflict misses in sequential codes. When we examine the cache behavior of parallel codes, we see that similar problems exist. Figure 5 presents a parallel code, where iterations of the `doall` loop are executed in parallel by multiple processors. The iterations are usually divided into simple chunks between processors. For instance, with four processors the first processor (`id` = 0) would execute iterations 1 through N/4. The second processor (`id` = 1) would execute iterations N/4+1 to N/2 . We see that despite having its iterations divided between processors, the conditions resulting in conflict misses are quite

```
real S, A(N), B(N) -> real S, A(N), DUM(PAD), B(N)
doall i=1,N           do i = id*N/P+1,(id+1)*N/P
  S = S+A(i)*B(i)        S = S+A(i)*B(i)
```

Figure 5: Padding for Parallel Code

similar, if not identical, to the conditions found in sequential programs. As a result, we expect data layout optimizations to be also effective in eliminating conflict misses on parallel programs.

## 4 Experimental Evaluation

### 4.1 Evaluation Framework

We implemented our transformations in the Stanford SUIF compiler [22] and used it to optimize kernels and two applications from SPEC95 as shown in Table 1. The original and padded versions of each program were both timed and simulated for a number of cache configurations using SHADE from Sun Microsystems. Our *base* cache configuration was a 16K direct-mapped cache with 32B lines. The resulting observations demonstrate the potential effectiveness of our transformations. Only one heuristic, PAD, was applied to CHOL and DGEFA since these kernels only possess one array. GROUPPAD was not applied to ERLE because our prototype compiler is currently unable to pad 3D arrays.

### 4.2 Compile-Time Statistics

Table 2 illustrates the effectiveness of GROUPPAD in preserving group reuse when targeting a range of different cache sizes. For these statistics, we consider only group reuse across outer loop iterations, since these are the most difficult to exploit. For instance, `A(j,i+1)` and `A(j,i)` would be counted as reuse while `A(j+1,i)` and `A(j,i)` would not.

INHERENT REUSE contains the total number of instances of group reuse for each program. Given a large enough cache, the data layout for any program can be altered to preserve all of its inherent group reuse. REUSE OBTAINED gives the number of group reuses preserved by GROUPPAD for the various cache sizes. For the larger caches (32K, 64K) GROUPPAD is able to exploit all group reuse, while for smaller caches (4K, 8K), almost no reuse can be preserved. Further examination of the access patterns in these programs has indicated that for the 16K cache, GROUPPAD obtains maximal group reuse, the highest attainable by any program transformation, for all programs except EXPL, where one additional reuse can be preserved.

Table 3 compares the total amount of memory used by PAD, MAXPAD, and GROUPPAD for padding. MAXPADLITE data is omitted since it produces the same layout as MAXPAD for the base cache. These figures indicate that while all methods result in small increases in memory size ($< 1\%$) relative to the data set size, PAD is far more economical than MAXPAD and GROUPPAD, and GROUPPAD uses the most memory.

### 4.3 Impact on Cache Miss Rates

To examine the effectiveness of our padding heuristics, we created many versions of our benchmark programs and simulated their miss rates on our base cache. Figure 6 presents cache miss rates obtained. Programs are listed along the X-axis, miss rates (in %) are shown along the Y-axis. The original versions of each program experience high conflict

| Program | Lines | Description |
|---|---|---|
| adi32 | 63 | 2D ADI Integration Fragment (Liv8) |
| chol256 | 90 | Cholesky Factorization |
| dgefa256 | 72 | Gaussian Elimination with Pivoting |
| dot256 | 32 | Vector Dot Product (Liv3) |
| erle64 | 612 | 3D Tridiagonal Solver |
| expl512 | 59 | 2D Explicit Hydrodynamics (Liv18) |
| jacobi512 | 52 | 2D Jacobi with Convergence Test |
| shal512 | 227 | Shallow Water Model |
| swim | 487 | Vector Shallow Water Model |
| tomcatv | 195 | Mesh Generation |

Table 1: Test Programs for Experiments

| | Inherent | Reuse Obtained | | | | |
|---|---|---|---|---|---|---|
| Program | Reuse | 4k | 8k | 16k | 32k | 64k |
| expl512 | 9 | 0 | 0 | 3 | 9 | 9 |
| jacobi512 | 2 | 0 | 0 | 2 | 2 | 2 |
| shal512 | 8 | 0 | 1 | 6 | 8 | 8 |
| swim | 8 | 1 | 6 | 8 | 8 | 8 |
| tomcatv | 10 | 0 | 0 | 6 | 10 | 10 |

Table 2: GROUPPAD Group Reuse Measures

| | Data | Data Size Increase | | | | | |
|---|---|---|---|---|---|---|---|
| | Set | PAD | | MAXPAD | | GROUPPAD | |
| Program | Mb | byte | % | Kb | % | Kb | % |
| expl512 | 18 | 304 | 0.00 | 14 | 0.08 | 8 | 0.05 |
| jacobi512 | 4 | 40 | 0.00 | 8 | 0.20 | 4 | 0.10 |
| shal512 | 28 | 448 | 0.00 | 15 | 0.05 | 16 | 0.06 |
| swim | 14 | 48 | 0.00 | 9 | 0.06 | 110 | 0.77 |
| tomcatv | 14 | 96 | 0.00 | 7 | 0.05 | 36 | 0.25 |

Table 3: Comparison of Padding Required by Heuristics

miss rates. All padding heuristics were successful in eliminating severe conflict misses and significantly reducing miss rates.

To compare padding heuristics in greater detail, we measured miss rates for a direct-mapped cache for a range of cache sizes (4K–64K). Miss rate improvements relative to PAD were computed for MAXPADLITE, MAXPAD, and GROUPPAD by subtracting the miss rate for PAD from the miss rate for each heuristic. Results are shown in Figure 7.

We see that MAXPADLITE fails to eliminate conflict misses in two cases (JACOBI, 8K cache and SWIM, 4k cache). Otherwise the miss rates for MAXPADLITE are similar to those for MAXPAD. By avoiding severe conflict misses, MAXPAD is able to avoid the degradations found in MAXPADLITE. Both heuristics reduce miss rates for EXPL at 64K by 5%, JACOBI at 16-64K by 2%, and TOMCATV at 16K by 2%. Smaller improvements are found for ERLE, SHAL, and SWIM. GROUPPAD achieves the same improvements as MAXPAD, but in addition manages to improve EXPL and SHAL at 16-32K due to its more precise analysis of group reuse. GROUPPAD slightly improves SWIM at 8–16K, but degrades TOMCATV at 64K.

### 4.4 Effect on Sequential Execution Time

We measured sequential execution time on the DEC Alpha 21064, Sun UltraSparc2, and Intel Pentium2. Results are presented in Figure 8. Programs are listed on the X-axis, improvement (in %) relative to the execution time of the original program is shown on the Y-axis. On the Alpha and Sparc2 (both with direct-mapped caches) all padding heuristics improve performance by 10% or more for all programs. On the Pentium2, performance of four programs (ADI, DOT,

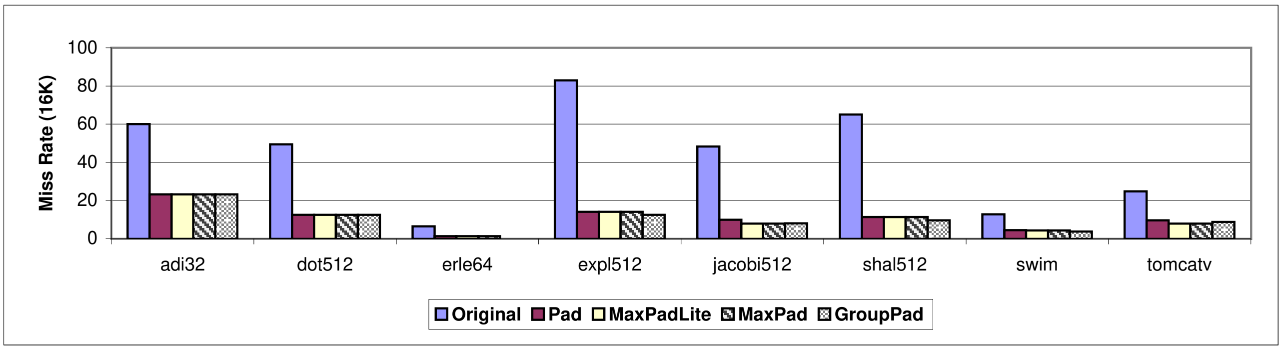


Figure 6: Cache Miss Rates

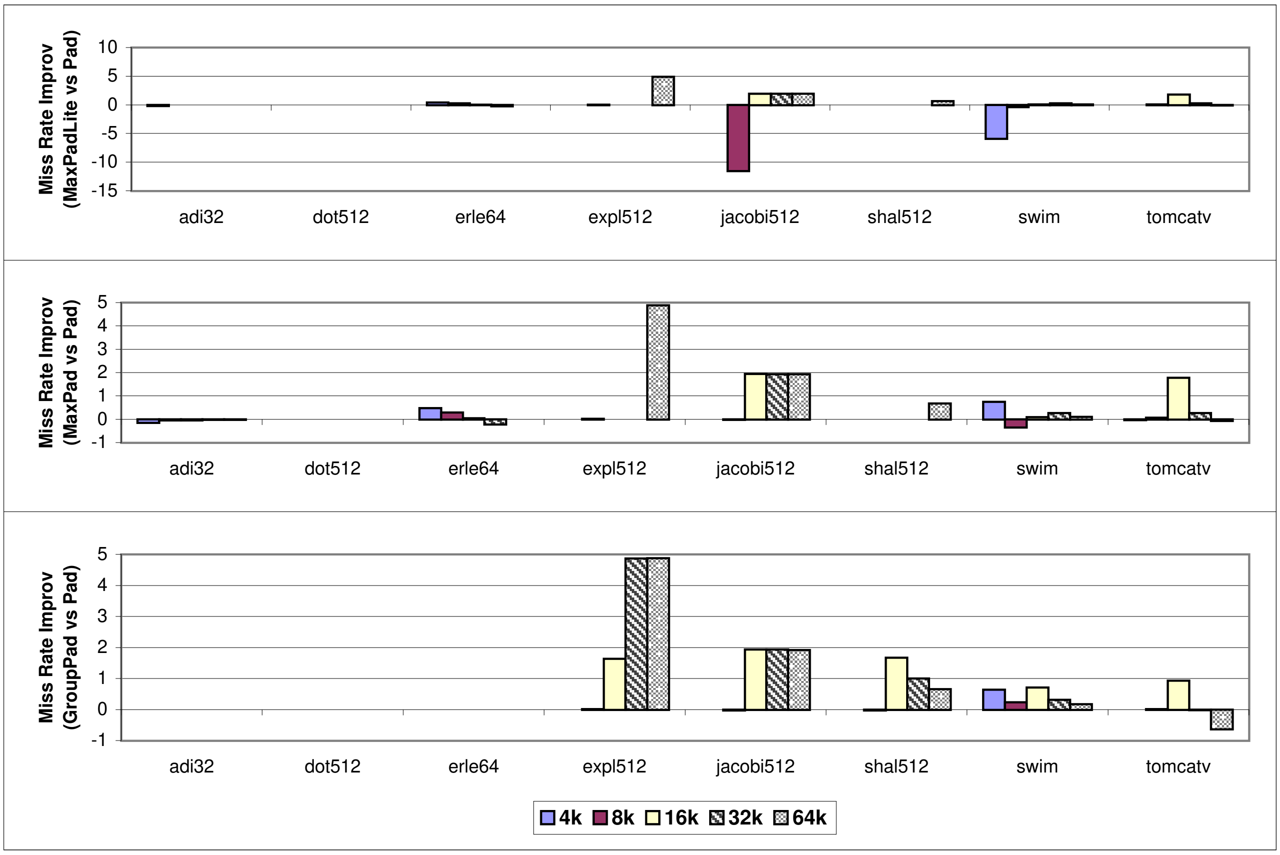


Figure 7: Miss Rates Improvements With Varying Cache Sizes

ERLE, JACOBI) are barely affected. These programs have conflicts from a small number of arrays and are resolved by the Pentium2's higher associativity cache. However, the remaining programs still benefit from padding.

In most cases MAXPADLITE, MAXPAD, and GROUPPAD yield the same improvements as PAD. MAXPADLITE and MAXPAD do better for DOT on the Sparc2. MAXPADLITE, MAXPAD, and GROUPPAD are better for EXPL on the Alpha and Pentium2, and SHAL, TOMCATV on the Sparc2. GROUPPAD is significantly better for JACOBI on the Alpha and Pentium2. These improvements were not predicted by our primary cache simulations, and may be caused by the secondary cache or other parts of the memory hierarchy. We are conducting additional research to explain these results.

### 4.5 Effect on Parallel Execution Time

As we discussed earlier, the structure of parallel programs is sufficiently similar to sequential versions that we expect our optimizations to be also successful at eliminating conflict misses in parallel programs. We used SUIF to generate parallel programs for two systems, Pthreads on a 4-processor Alpha workstation (shared-memory) and CVM [13] on an 8-processor IBM SP-2 (software DSM). Results are displayed in Figure 8. Programs are listed on the X-axis, improvement (in %) relative to the execution time of the original parallel program is shown on the Y-axis. We see padding transformations do improve parallel performance for both systems. The magnitude of the improvements are also similar to the improvements in sequential execution times.

Recently, compiler-directed *page coloring* was proposed

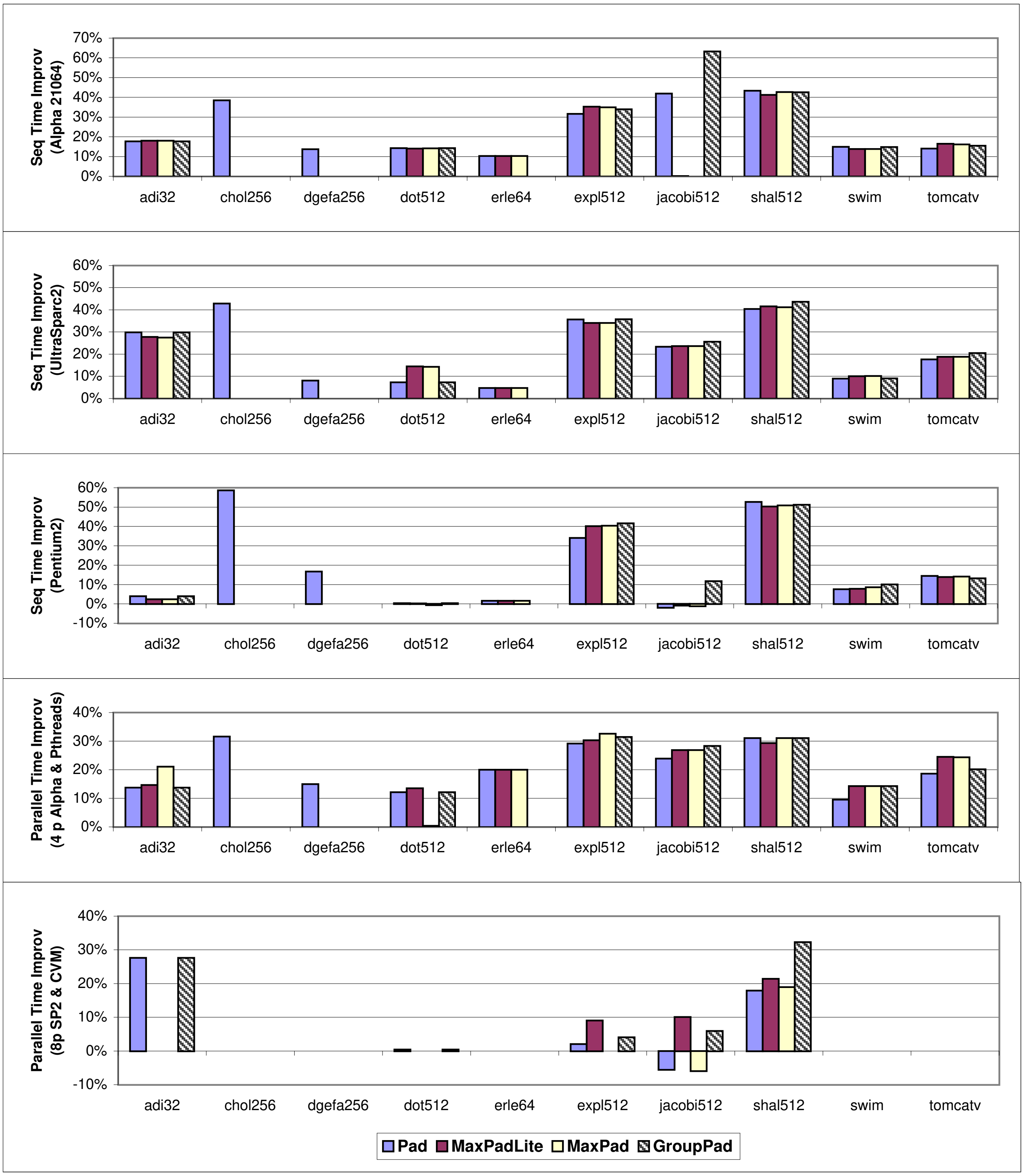


Figure 8: Execution Time Improvements (Alpha, UltraSparc2, Pentium2, Pthreads, CVM)

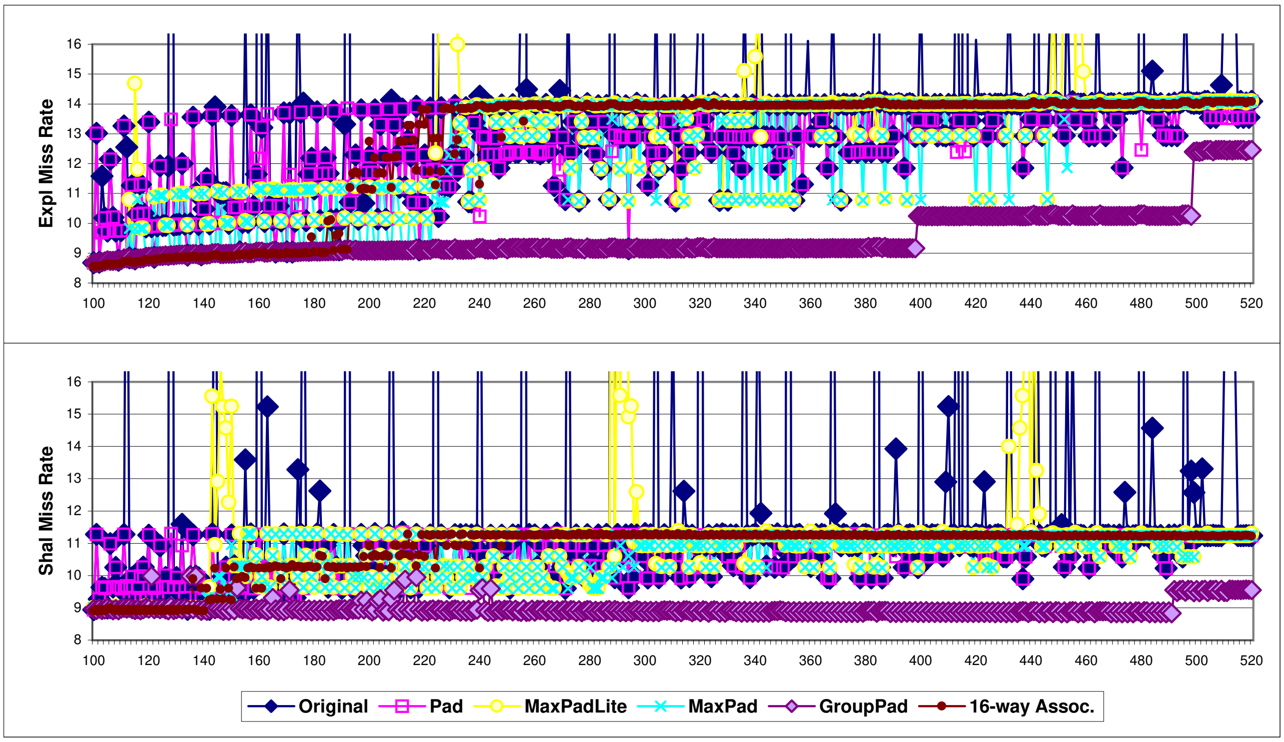


Figure 9: Cache Miss Rates (Varying Problem Size)

as a method for reducing conflict misses in multiprocessors [3]. Two of the SPEC95 programs which benefited the most were SWIM and TOMCATV. We find in our experiments that both also benefit from inter-variable padding. It is possible that for stencil computations, compile-time padding could achieve many of the same benefits as page coloring.

### 4.6 Varying Problem Sizes

To compare the PAD, MAXPADLITE, MAXPAD, and GROUPPAD heuristics more thoroughly, we varied the problem sizes for two kernels, EXPL and SHAL, from 100 to 520, and simulated miss rates for each version on the base cache. Additionally, we simulated miss rates for the original program on a 16K 16-way associative caches. Results are shown in Figure 9. The X-axis represents problem sizes, while the Y-axis presents the cache miss rate.

We see the original versions of each kernel experience severe conflict misses for some problem sizes, particularly powers of two. In fact, we had to cut off some of the higher miss rates to make the remainder of the graph legible. PAD eliminates these severe conflicts, but miss rates still fluctuate over a range of about 3% for EXPL and 2% for SHAL. MAXPADLITE eliminates the original severe conflicts and improves on PAD in several cases, but experiences severe conflicts for several clusters of problem sizes, when pads are near array column sizes. MAXPAD is able to avoid the new severe conflicts introduced by MAXPADLITE. Both MAXPADLITE and MAXPAD still experience fluctuations in the miss rate, as different amounts of group reuse are exploited.

In comparison, GROUPPAD obtains miss rates consistently better than the other heuristics, by about 5% for EXPL and over 2% for SHAL on average. In fact, GROUPPAD consistently yields miss rates several percent lower than the original program on a 16-way associative cache. In addition, the miss rates for GROUPPAD usually deviate only by a few hundredths of a percentage. It is instructive to note that cache miss rates for GROUPPAD generally increase according to a step function. This effect is caused by the data size increasing to the point where the number of group reuses which the compiler can preserve in cache goes down by one. The only fluctuations for GROUPPAD occur for some small sizes of SHAL, and are caused by the compiler trying to tweak the data layout to avoid severe conflict misses for inexpensive loops. Overall, results indicate GROUPPAD is quite robust in its ability to choose paddings to exploit group reuse.

## 5 Related Work

Data locality has been recognized as a significant performance issue for both scalar and parallel architectures. In particular, conflict misses can cause half of all cache misses and most intra-nest misses in scientific codes [18]. Conflicts may be eliminated with hardware [9, 11] or operating systems support [2, 3]. For many scientific codes we can achieve similar or better results through inexpensive data layout transformations. Data-layout transformations applied by hand has been shown to reduce conflict misses in the SPEC benchmarks [15].

Researchers working on compile-time data locality optimizations have concentrated on computation-reordering such as loop permutation [7, 17, 23], and tiling [5, 14, 23], though loop fission and fusion have also been found to be helpful [16, 17]. Many cache models have been designed for estimating cache misses to help guide data locality optimizations [6, 7, 17, 23]. These models typically can predict only capacity misses because they assume a fully-associative cache. Temam *et al.* present a method for detecting and counting the number of conflict misses [21]. Ghosh *et al.* calculate conflict misses accurately by counting the number of integer solutions to *cache miss equations*, linear Diophantine equations that exactly specify the cache line to which each reference is mapped for every loop [8]. Our heuristics for detecting conflict misses are less accurate, but are inexpensive and allow us to optimize larger programs.

Researchers have examined data layout transformations, usually in the context of parallel programs. Techniques include changing the placement of variables in explicitly parallel programs [10], reshaping arrays to make local data contiguous [1], and combining array transpose with loop permutation to improve parallelism and locality [4, 12, 19]. Manjikian and Abdelrahman perform *cache partitioning*, spacing out variables as far as possible in a cache, in order to reduce conflict misses in parallel programs after loop fusion [16]. We improve on their algorithm in this paper. In previous research, we developed padding techniques for eliminating severe cache conflicts in stencils and linear algebra computations [20]. In this paper we improve our heuristic to preserve group reuse across outer loop iterations, as well as evaluate its impact on parallel applications.

## 6 Conclusions

Padding transformations have been proven useful for eliminating cache conflict misses. In this paper, we evaluate GroupPad, a new inter-variable padding technique for preserving group reuse. GroupPad is applicable only to stencil computations; more research is needed to extend it to a wider range of programs. We also establish the usefulness of padding for eliminating conflict misses in parallel programs. The magnitude of improvements appears similar to that observed in padding sequential versions of the same program. Our prototype compiler achieves respectable improvements on average and significant improvements for some programs. Results indicate padding transformations can be very useful for both sequential and parallel machines.

## References

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

[2] B. Bershad, D. Lee, T. Romer, and B. Chen. Avoiding conflict misses dynamically in large direct-mapped caches. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, San Jose, CA, October 1994.

[3] 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.

[4] 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.

[5] S. Coleman and K. S. 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.

[6] 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.

[7] 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.

[8] 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.

[9] A. Gonzalez, M. Valero, N. Topham, and J. Parcerisa. Eliminating cache conflict misses through XOR-based placement functions. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, Vienna, Austria, July 1997.

[10] 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.

[11] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th International Symposium on Computer Architecture*, Seattle, WA, May 1990.

[12] M. Kandemir, J. Ramanujam, and A. Choudhary. A compiler algorithm for optimizing locality in loop nests. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, Vienna, Austria, July 1997.

[13] 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.

[14] 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.

[15] A. Lebeck and D. Wood. Cache profining and the SPEC benchmarks: A case study. *IEEE Computer*, 27(10):15–26, October 1994.

[16] N. Manjikian and T. Abdelrahman. Fusion of loops for parallelism and locality. *IEEE Transactions on Parallel and Distributed Systems*, 8(2):193–209, February 1997.

[17] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

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

[19] M. O'Boyle and P. Knijnenburg. Non-singular data transformations: Definition, validity, and applications. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, Vienna, Austria, July 1997.

[20] G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, Canada, July 1998.

[21] O. Temam, C. Fricker, and W. Jalby. Cache interference phenomena. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*, Santa Clara, CA, May 1994.

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

[23] 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.