Data Transformations for Eliminating Conflict Misses

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. We examine two compile-time data-layout transformations for eliminating conflict misses, concentrating on misses occuring on every loop iteration. Inter-variable padding adjusts variable base addresses, while intra-variable padding modifies array dimension sizes. Two levels of precision are evaluated. PADLITE only uses array and column dimension sizes, relying on assumptions about common array reference patterns. PAD analyzes programs, detecting conflict misses by linearizing array references and calculating *conflict distances* between uniformly-generated references. The Euclidean algorithm for computing the *gcd* of two numbers is used to predict conflicts between different array columns for linear algebra codes. Experiments on a range of programs indicate PADLITE can eliminate conflicts for benchmarks, but PAD is more effective over a range of cache and problem sizes. Padding reduces cache miss rates by 16% on average for a 16K direct-mapped cache. Execution times are reduced by 6% on average, with some SPEC95 programs improving up to 15%.

1 Introduction

Because of the increasing gap between processor and memory speeds, programs can achieve high performance only if they use caches effectively. 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 believe compiler transformations can be very effective in eliminating conflict misses for scientific programs with regular access patterns. We evaluate two compiler transformations to eliminate conflict misses: inter- and intra-variable padding [3, 21]. Unlike standard compiler transformations which restructure the computation performed by the program, these two techniques modify the program's *data layout*. Both padding transformations have been discussed in the literature, but we are the first to: 1) develop and implement efficient, practical compile-time padding heuristics, 2) empirically evaluate padding transformations over a large collection of programs, and 3) investigate precision of analysis needed to guide padding for different cache and problem sizes.

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

In Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98), Montreal, Canada, June 1998.

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

Figure 1: Inter-Variable Padding (Change Base Addr)

```
real A(N,N), B(N,N) -> real A(N+PAD,N), B(N,N)
do i = 2,N-1
 do j = 2,N-1
  B(j,i) = (A(j-1,i)+A(j,i-1)+A(j+1,i)+A(j,i+1))/4
```

Figure 2: Intra-Variable Padding (Change Dim. Size)

1.1 Severe Conflicts

We begin by examining two motivating examples for padding transformations. In Figure 1, unit-stride references to `A(i)` and `B(i)` provide spatial locality, leading to cache reuse. However, if `A` and `B` are separated by a multiple of the cache size and the cache is direct mapped, every access to `B(i)` will map to the same cache line as the previous `A(i)` access and vice versa. As a result, every reference will produce a conflict miss. To regain the benefits of spatial reuse, inter-variable padding may be applied as shown in Figure 1.

In Figure 2, the stencil computation exhibits much spatial and temporal reuse between the references to `A`. However, if the column size of `A` is a multiple of the cache size, columns of `A` will conflict, eliminating reuse between references to `A`. To regain spatial reuse, intra-variable padding applied to the columns of `A` can change its internal layout so that columns no longer map to conflicting locations on the cache, as shown in Figure 2.

Both of these examples show instances of *severe* conflict misses, when references cause cache lines to be flushed to memory on every loop iteration. In linear algebra kernels, *semi-severe* conflicts may exist which occur regularly on a large percentage of loop iterations. In this paper, we develop and evaluate several heuristics for applying data layout transformations to eliminate these severe conflict misses.

2 Padding Heuristics

We present PADLITE and PAD, two heuristics for guiding both inter- and intra-variable padding. PADLITE only requires knowledge of array dimension sizes to choose pads. To replace analysis, PADLITE relies on assumptions about data access patterns found in real programs. In comparison, PAD analyzes array references in the program to choose pads. PAD is more precise, at the expense of greater complexity.

For convenience, we refer to the inter-variable padding portion of PADLITE and PAD as INTERPADLITE and INTERPAD, respectively. Similarly, the intra-variable padding portion of PADLITE and PAD are refered to as INTRAPADLITE and INTRAPAD, respectively. We also present LINPAD1 and LINPAD2, two intra-variable padding heuristics for eliminating conflict misses in linear algebra codes that are applicable to both PADLITE and PAD. We describe each padding heuristic separately, then show how they are combined to form PADLITE and PAD. For each heuristic, we present a *pad condition* which when true indicates padding should be applied.

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.

2.1 Inter-variable Padding

We begin by describing INTERPADLITE and INTERPAD. The analysis required by INTERPADLITE is sufficiently simple that it can be conveniently performed by the linker, making INTERPADLITE a good heuristic for performing inter-variable padding at link time.

2.1.1 INTERPADLITE

Motivated by the assumption that severe conflicts commonly arise from interference between references to variables of the same size, INTERPADLITE attempts to separate base addresses of equally-size variables from one another on the cache. While separation by one cache line will ensure that references of the form `B(i)` and `C(i)` do not conflict, a larger separation may be required for references such as `B(i)` and `C(i-2)`. Therefore, a value M, typically a small number of cache lines, is chosen as a *minimum separation distance* for these base addresses. In Section 4.3, we give results supporting the choice of $M = 4$ cache lines. This value is adopted for M in the remaining discussion.

Padding is performed in a simple, greedy fashion. Variable base addresses are initially undefined and then assigned one at a time. The first variable is assigned to location 0. The next available location, 0 + the size of the first variable, is considered as a tentative location for the second variable. Similarly, on each iteration, the next available location is considered as a tentative location for the next variable. The tentative location may be incremented, or *padded*, if it is not acceptable.

A variable's tentative location is rejected when it has a conflict distance less than M from some other equally-sized variable whose base address has been already assigned. When this pad condition occurs, the tentative location is incremented as necessary. This test-and-pad step is repeated, ending as soon as a location suitably spaced from all equally-sized variables has been found. In the event that the location is incremented beyond its original position by a distance larger than the cache size, no satisfactory base address is possible and the initial tentative location is assigned.

For this case to occur, a base address must map to each block of size $2M$ on the cache. Satisfactory base addresses will be found for all variables therefore when the number of variables of the same size does not exceed $C_s/(2M)$ (while INTERPADLITE can successfully lay out C_s/M such variables when base addresses are placed multiples of M apart from one other, less regular spacings may result, permitting the assignment of fewer variables). For example, if $C_s = 2048$ and $M = 16$, up to 64 equally-sized variables are guaranteed to be padded a distance at least M from one another. In our experiments on a 16K cache, INTERPADLITE has always found a satisfactory layout.

2.1.2 INTERPAD

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

We can calculate the memory address of a multidimensional array reference by linearizing its subscripts. Subtracting two linearized references gives their distance. When the distance is computed between any two uniformly generated references on a given loop iteration, we obtain an expression in which all i_j terms cancel:

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

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

Base addresses are selected by INTERPAD using a greedy approach similar to INTERPADLITE. Variables begin with undefined base addresses which are assigned in turn. For each variable `A`, we first consider the next available base address. For all variables `B` with defined base addresses, we compute the conflict distances between the uniformly generated references of `A` and `B` over all loops. If any distance in any loop $< L_s$, the tentative location is incremented as necessary to eliminate this pad condition and the test repeats. Observe that making the conflict distance L_s or greater is a sufficient (though not necessary) condition for eliminating severe conflicts between two references. As with PADLITE, if this location is incremented beyond its initial position by a distance larger than the cache size, the heuristic has failed to find a satisfactory address and defaults to the original tentative location. In our experiments PAD has always found a non-conflicting base address.

We should point out that this technique can easily be generalized for multilevel caches. The only modification is to compute conflict distances with respect to each cache configuration and then to pad as needed if any distance is less than the corresponding cache line size. The same is true for IntraPad, described later.

2.2 Intra-variable Padding for Stencil Codes

The next two sections describe heuristics for IntraPadLite and IntraPad, the intra-variable padding portion of PadLite and Pad. These heuristics are designed to eliminate severe conflicts commonly occurring in stencil computations. Additional intra-variable padding heuristics are described later in Section 2.3. IntraPadLite pads arrays based only on array dimension sizes. IntraPad again applies analysis of uniformly generated references.

2.2.1 IntraPadLite

IntraPadLite continues using the parameter M, now as the minimum distance of separation between nearby *columns* (or, for arrays with dimensionality 3 or higher, *subarrays*). For example, given a large enough M, maintaining a separation of M on the cache between the first elements of adjacent columns will ensure that references such as `B(i,j-1)` and `B(i+1,j)` do not conflict. For each array, IntraPadLite determines whether Col_s or $2Col_s$ are within M of a multiple of C_s (whether either Col_s or $2Col_s$ have conflict distances $< M$ with 0.) If so, the column size is increased as needed to eliminate this pad condition. Provided $C_s > 3M$, a pad of $2M$ (in units of array elements) or less is always sufficient. IntraPadLite pads arrays of higher dimensionality in a similar manner. When 1 or 2 times the size of any subarray is within M of a multiple of C_s, lower dimension sizes are increased so that this condition is no longer true.

2.2.2 IntraPad

IntraPad again considers uniformly generated references, this time to find references to the *same* array accessing memory locations apart by a fixed distance. By the definition given earlier, two references to the same array are uniformly generated as long as they are of the form $A(i_1 + r_1, i_2 + r_2, ..., i_d + r_d)$, $A(i_1 + s_1, i_2 + s_2, ..., i_d + s_d)$, again where each i_j is an unmodified variable or 0, and each r_j and s_j is an arbitrary integer. All other conditions are always satisfied since the two references access the same array. Expression 1 in this special case reduces to

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

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

For each array, the heuristic considers the uniformly generated references to only that array in each loop. If the conflict distance between a pair of references in any loop is less than L_s, a pad of one element is attempted on each of the lower dimensions in turn until this pad condition is no longer true. An upper bound on pad size is imposed to ensure termination. However, small pads have sufficed in practice. In our experiments on an 16K cache, arrays usually require pads of 2 or fewer elements, padding at most 3 elements.

```
do k
  do j
    do i
      A(i,j), A(i,k)
```

Figure 3: Simplified Linear Algebra Computation

2.3 Intra-variable Padding for Linear Algebra Codes

Memory access patterns common in linear algebra computations may also lead to frequent conflict misses. Consider the simplified loop nest shown in Figure 3. As the values of `j` and `k` vary, there exist column sizes of `A` (*e.g.,* 256, 384, 512) which lead to frequent conflict misses. Though these *semi-severe* conflict misses do not occur on every loop iteration, they may occur in a high percentage of these iterations, considerably degrading performance.

To eliminate these conflicts, we develop two intra-variable padding heuristics, LinPad1 and LinPad2, designed for linear algebra computations. LinPad2 is more comprehensive than LinPad1, but as a result is more aggressive in padding. Either heuristic may be incorporated into IntraPadLite or IntraPad, increasing the likelihood that the transformations will pad a given array in any program. Section 2.4 discusses how these padding heuristics may be combined.

2.3.1 LinPad1

LinPad1 avoids column sizes with large powers of two as factors. In particular, the pad condition holds when the column size is evenly divided by $2L_s$. Since C_s is a power of two, any column size which is a multiple of a large power of two will share a large greatest common divisor (gcd) with the cache size. Thus, multiples of the column size will thus map to a small number of locations on the cache. Consider Figure 3, given C_s and Col_s (both in units of array elements). If we let $d = gcd(C_s, Col_s)$, it follows that $(C_s/d)Col_s \equiv 0 \pmod{C_s}$, implying only the first C_s/d multiples of Col_s will map to distinct locations on the cache. This can lead to excessive conflict misses when d is large. For example, supposing $C_s = 1024$ and $Col_s = 768$, we find $d = gcd(C_s, Col_s) = 256$. Thus, $4 \times Col_s \equiv 0 \pmod{C_s}$, meaning any two values $jCol_s$ and $kCol_s$ will have a conflict distance of 0 when $j - k$ is a multiple of 4. Under this column size, the references in Figure 3 will conflict in approximately 1/4 of the iterations of the `j` loop. LinPad1 thus pads array columns as necessary to avoid the pad condition.

2.3.2 LinPad2

LinPad2 avoids column sizes Col_s where for small values of j, $jCol_s$ is within a cache line (L_s) of a multiple of C_s. For instance, given $C_s = 1024$, $Col_s = 273$, and $L_s = 4$, we reject Col_s since $15 \times 273 \equiv -1 \pmod{C_s}$. It also follows that $30 \times 273 \equiv -2 \pmod{C_s}$ and $45 \times 273 \equiv -3 \pmod{C_s}$, meaning columns apart by either 15, 30, or 45 may conflict. The LinPad2 pad condition, that columns j apart have a conflict distance less than L_s, subsumes the LinPad1 pad condition where the conflict distance must be zero.

More formally, given Col_s, we first define a *conflicting* j value as one for which $jCol_s$ is within L_s of a multiple of C_s. Next, as described below, we compute a parameter j^* as the the smallest tolerable conflicting j value. Finally,

$FirstConflict*(r, r', c, c', L_s)$
  if $r' < L_s$ then return c'
  else return $FirstConflict*(r', r \bmod r', c', \lfloor r/r' \rfloor c' + c, L_s)$

$FirstConflict(C_s, Col_s, L_s)$
  return $FirstConflict*(C_s, Col_s, 0, 1, L_s)$

Figure 4: Functions for Computing *FirstConflict*

we decide to reject Col_s if there exists a conflicting j value such that $0 < j < j^*$. This requires only finding the smallest positive conflicting j value since larger conflicting j values are irrelevant. The function shown in Figure 4 performs this task. *FirstConflict* makes the initial call to a recursive auxiliary function *FirstConflict**. The latter is a generalization of the Euclidean algorithm for computing the greatest common denominator. At each call, the invariant is maintained that (mod C_s) either $(c'Col_s \equiv r')$ or $(c'Col_s \equiv -r')$, and that for $0 < n < c'$, $j = n$ is not conflicting with respect to parameter L_s. Since successive r' values form the sequence of remainders computed by the Euclidean algorithm, r' decreases at each recurrence. Thus, we need only compare r' to L_s, returning c' as the smallest conflicting value once r' is smaller. Coleman and McKinley use a related algorithm for finding non-conflicting tile sizes [7].

The value j^* is selected to meet three criteria. First, j^* must be small enough to avoid excessive or infinite testing of successively larger column sizes. This is needed since transformations using LinPad2 iteratively test larger and larger column sizes until the heuristic decides not to reject. We accomplish this by limiting j^* to C_s/L_s, since any Col_s for which $gcd(Col_s, C_s) = L_s$ has a *FirstConflict* value of C_s/L_s. For instance, if $C_s = 1024$ and $L_s = 4$, then given a Col_s where $gcd(Col_s, C_s) = 4$, or equivalently, a Col_s which is a multiple of 4 but not of 8, it holds that $jCol_s$ is at least 4 or at most -4 (mod C_s) when $0 < j < 256 = C_s/L_s$ and that $256 Col_s \equiv 0$ (mod C_s). Thus, $FirstConflict(1024, Col_s, 4) = 256$. It follows that with $j^* \leq 256$, the search for a non-conflicting column size will terminate within 8 iterations, since every sequence of 8 consecutive integers contains a value with a *gcd* of 4 with C_s, and thus a *FirstConflict* value at least j^*. In general, $2L_s$ iterations are always sufficient when $j^* \leq C_s/L_s$.

Second, j^* should be large enough to substantially reduce conflict misses in linear algebra computations. We have determined experimentally that over a set of programs with arrays of varying sizes, the overall impact of padding improves with increases in j^* until about 129, after which benefits diminish. Finally, we impose another upper bound on the value of j^* based on the *number* of columns. Since accesses to columns j apart do not occur when there are fewer than j columns, j^* need not be greater than the row size (R_s). Thus, LinPad2 uses a j^* equal to 129 unless either ceiling reduces this amount. To summarize:

$$j^* = \min(129, R_s, C_s/L_s)$$

2.4 Combined Padding Heuristics

Now that we have presented each padding heuristic, we combine them into the actual PadLite and Pad algorithms. Both PadLite and Pad consist of first applying intra-variable padding, then applying inter-variable padding. Intra-variable padding must be performed first, because it changes array sizes and thus variable base addresses.

Heuristic	Analyses Needed	Pad Condition
InterPadLite	var size	Δ base addrs $< M$
IntraPadLite	var size+dims	$\Delta[1..2]Col_s < M$
InterPad IntraPad	var size+dims +subscripts	Δ ref addrs $< L_s$
LinPad1 LinPad2	var size+dims	$\Delta[1..j]Col_s = 0$ $\Delta[1..j]Col_s < L_s$

Table 1: Properties for Padding Heuristics

2.5 PadLite

PadLite first gathers information on array and dimension sizes. Next it performs intra-variable padding, combining IntraPadLite and LinPad1 by testing both pad conditions, trying larger pads when either condition is true. It then applies inter-variable padding with InterPadLite. We chose LinPad1 because PadLite is unable to recognize linear algebra codes and will apply intra-array padding indiscriminately. LinPad1 is less aggressive than LinPad2, and is thus less likely to introduce unnecessary pads.

2.6 Pad

Pad first gathers information on array and dimension sizes, then extracts uniformly-generated references from each loop nest. Next it performs intra-variable padding, combining IntraPad and LinPad2 by testing both pad conditions, trying larger pads when either condition is true. Since Pad may analyze array references, it tests LinPad2 only for arrays appearing in computations of the form shown in Figure 3. Pad then applies inter-variable padding with InterPad. We chose LinPad2 because Pad is able to apply it only to arrays appearing in linear algebra codes. In Section 4.5 we present data to justify our choice of intra-variable padding heuristics for PadLite and Pad.

2.7 Summary of Heuristics

Table 1 summarizes the required analyses and pad conditions for all heuristics. From the column Analyses Needed, we see InterPadLite uses only variable sizes and that IntraPadLite inspects array dimensions in addition. LinPad1 and LinPad2 require only column sizes as well. InterPad and IntraPad analyze array subscripts to compute exact conflict distances.

The column Pad Condition indicates how each heuristic decides to advance base addresses or increase column sizes. This decision generally involves computing conflict distances between various quantities. InterPadLite pads when the conflict distance between base addresses is less than the minimum distance M. IntraPadLite pads when the conflict distance for either Col_s or $2Col_s$ is less than M. InterPad and IntraPad both compute conflict distances of linearized array references, padding when it is less than the cache line size L_s. LinPad1 pads when two nearby columns have a conflict distance of zero. LinPad2 pads when two nearby columns have a conflict distance less than L_s.

Figure 5 summarizes the general structure of both InterPadLite and InterPad. Both heuristics maintain a tentative base address *addr* which is padded in the presense of conflicts. The decision of whether and how much to pad is abstracted by the *neededPad* function and relates to the pad conditions shown in Table 1. For instance, InterPadLite determines how much padding is needed by

$BaseAddress(A) \leftarrow NULL\ \forall$ vars A
$addr \leftarrow 0$
foreach var A
  repeat
    $padsize \leftarrow 0$
    foreach var B where $BaseAddress(B) \neq NULL$
      $padsize \leftarrow \max(padsize, neededPad(A, addr, B))$
    if $padsize > 0$ then $addr \leftarrow addr + padsize$
  until $padsize = 0$
  $BaseAddress(A) \leftarrow addr$
  $addr \leftarrow addr + sizeOf(A)$

Figure 5: General InterPadLite and InterPad Algorithm

foreach var A
  repeat
    $padsize \leftarrow \max(neededStencilPad(A),$
        $neededLinAlgPad(A))$
    if $padsize > 0$ then increase $Col_s(A)$ by $padsize$
  until $padsize = 0$

Figure 6: General IntraPadLite and IntraPad Algorithm

computing which increment to *addr* is required to yield a conflict distance $< M$ between *addr* and $BaseAddr(B)$.

Figure 6 summarizes the general structure of intra-variable padding in both PadLite and Pad. The function *neededStencilPad* based on the IntraPadLite or IntraPad conditions of Table 1. The function *neededLinAlgPad* returns pad sizes uses the LinPad1 or LinPad2 conditions. A more general algorithm handles arrays of dimensionality higher than two.

3 Sample Transformations

Examining the application of PadLite and Pad will illustrate these heuristics and their limitations. Figure 7 expands on Figure 2, showing the key loop nests of jacobi, an iterative solver for partial differential equations (the convergence test is not shown). In this kernel, A and B are both $N \times N$ arrays. Our transformations assume a write-allocating/write-back cache, so any two accesses may conflict, whether write or read. We consider outcomes when altering the values of parameters N, C_s (the cache size), and L_s (the cache line size.) For simplicity, we assume only stencil intra-variable padding heuristics are used in the following examples. All values discussed are in units of jacobi array elements.

```
do i = 2,N-1
  do j = 2,N-1
    B(j,i) = 0.25 * (A(j-1,i) + A(j,i-1) +
                     A(j+1,i) + A(j,i+1))
do i = 2,N-1
  do j = 2,N-1
    A(j,i) = B(j,i)
```

Figure 7: Key Loop Nests in jacobi

$N = 512$, $C_S = 2048$, $L_S = 4$ PadLite begins with IntraPadLite, which finds N too small to induce intra-array padding. InterPadLite begins, putting A at location 0, making the tentative location for B $512 \times 512 \equiv 0$, mod C_s. B is therefore advanced by M. Pad begins with IntraPad. IntraPad finds that no A references conflict with one another and that B does not have multiple references in any loop. Therefore, the column sizes of A and B are unchanged. InterPad puts A at 0 and finds B references conflict in both loops. B's tentative location is therefore padded by 5. Thus, both heuristics perform adequately for these parameters. Inter-variable conflicts appear in both loops, and both Pad and PadLite are successful in eliminating them.

$N = 512$, $C_S = 1024$, $L_S = 4$ With $C_s = 1024$, IntraPadLite increments the column size of A since $2N$ mod C_s is 0. 8 pad elements are sufficient for $M = 16$. A's column size, and thus B's, are increased to 520. InterPadLite begins, placing A at 0. The tentative address for B is $520 \times 512 \equiv 0$, mod C_s. The sizes of A and B are still equal, so B is padded by M. IntraPad finds that references A(j,i-1) and A(j,i+1) have conflict distance 0. Padding A's column size by 2 eliminates all conflicts. InterPad places A at 0 and then places B immediately at 514×512, since A and B are no longer conforming. Therefore, under these parameters, both Pad and PadLite again perform adequately, eliminating all severe misses.

$N = 934$, $C_S = 1024$, $L_S = 4$ In this case, neither heuristic finds intra-variable padding necessary. InterPadLite applies no inter-variable padding as well since B at $934 \times 934 \equiv 932 \pmod{C_s}$ is sufficiently spaced from A. InterPad however computes a conflict distance of 2 between B(j,i) and A(j,i+1) since $934 \times 934 - 934 \equiv -2 \pmod{C_s}$ and pads B by 6 elements. PadLite therefore fails to eliminate the existing severe conflict misses. Analysis enables Pad to find a layout eliminating these conflicts.

4 Experimental Evaluation

4.1 Evaluation Framework

To experimentally evaluate inter- and intra-variable padding, we implemented both transformations in the Stanford SUIF compiler [22]. Additional transformations are performed to give the compiler control over variable base addresses. First, local array and structure variables are promoted into the global scope (globalization). Formal parameters to functions do not need to be promoted, since they represent variables declared elsewhere. Variables in Fortran common blocks are split into separate variables if permitted by *sequence association*. Otherwise they remain in a single common block and can not be padded. Global variables are then made into fields of a large structured variable, resulting in a single variable containing all of the variables to be optimized. The compiler can now modify variable base addresses by reordering fields in the structure and inserting pad variables. The compiler also determines whether intra-variable padding can be safely applied to each array, by determining whether the array violates *storage association* or is passed as a parameter to a procedure.

We used the SUIF compiler to optimize several scientific kernels and applications taken from the NAS, SPEC92, and SPEC95 benchmarks suites. Costs of applying Pad and PadLite were a very small percentage of overall compilation time. Various compile-time measures of the progress of our padding algorithms were collected. The original and padded versions of each program were then simulated for several cache configurations using Shade from Sun Microsystems. Our *base* cache configuration was a 16K direct-mapped cache with 32B lines. Where desired we simulated a 16-way associative cache in place of a fully-associative cache. We also timed the original and optimized versions of each program on a number of architectures. The resulting observations demonstrate the potential effectiveness of our transformations.

Program	Description	Lines	Global Arrays	% Unif. Refs	Arrays Safe	(Intra) Arrays Padded	Max # Incr	Total # Incr	(Inter) Bytes Skip	% Size Incr
				KERNELS						
ADI32	2D ADI Integration Fragment (Liv8)	63	6	100	6	3	2	6	64	0.0
CHOL256	Cholesky Factorization	165	5	100	4	1	1	1		0.2
DGEFA256	Gaussian Elimination w/Pivoting	75	2	100	2	1	1	1		0.4
DOT256	Vector Dot Product (Liv3)	32	2	100	2				32	0.0
ERLE64	3D Tridiagonal Solver	612	23	100	7	2	1	2	32	0.8
EXPL512	2D Explicit Hydrodynamics (Liv18)	64	9	100	9				304	0.0
IRR500K	Relaxation over Irregular Mesh	196	4	100	0					0.0
JACOBI512	2D Jacobi Iteration w/Convergence	52	2	100	2				40	0.0
LINPACKD	Gaussian Elimination w/Pivoting	795	6	99	1					0.0
MULT300	Matrix Multiplication (Liv21)	29	3	100	3					0.0
RB512	2D Red-Black Over-Relaxation	52	1	100	1					0.0
SHAL512	Shallow Water Model	235	15	100	14	9	3	20	384	0.3
SIMPLE	2D Hydrodynamics	1346	37	100	37					0.0
				NAS BENCHMARKS						
APPBT	Block-Tridiagonal PDE Solver	4441	42	95	26					0.0
APPLU	Parabolic/Elliptic PDE Solver	3417	34	100	17					0.0
APPSP	Scalar-Pentadiagonal PDE Solver	3991	41	87	23	1	1	1	384	0.4
BUK	Integer Bucket Sort	305	5	100	1				64	0.0
CGM	Sparse Conjugate Gradient	855	11	95	0				28	0.0
EMBAR	Monte Carlo	265	3	80	1					0.0
FFTPDE	3D Fast Fourier Transform	773	7	60	0				64	0.0
MGRID	Multigrid Solver	680	12	81	0					0.0
				SPEC95 BENCHMARKS						
APPLU	Parabolic/Elliptic PDE Solver	3868	33	100	17					0.0
APSI	Pseudospectral Air Pollution	7361	23	95	22					0.0
FPPPP	2 Electron Integral Derivative	2784	23	95	16					0.0
HYDRO2D	Navier-Stokes	4292	9	99	9				48	0.0
MGRID	Multigrid Solver	484	10	80	3					0.0
SU2COR	Quantum Physics	2332	117	93	71				24	0.0
SWIM	Vector Shallow Water Model	429	14	100	14				48	0.0
TOMCATV	Mesh Generation	190	9	100	9				96	0.0
TURB3D	Isotropic Turbulence	2100	38	94	22				352	0.0
WAVE5	Maxwell's Equations	7764	57	85	42					0.0
				SPEC92 BENCHMARKS						
DODUC	Thermohydraulical Modelization	5334	91	90	64					0.0
MDLJDP2	Molecular Dynamics (double prec)	4316	25	86	18					0.0
MDLJSP2	Molecular Dynamics (single prec)	3885	23	86	16					0.0
NASA7	NASA Ames Fortran Kernels	1204	38	95	36					0.0
ORA	Ray Tracing	453	0	100	0					0.0

Table 2: Compile-Time Statistics for PAD (16K direct-mapped cache, 32B lines)

4.2 Compile-Time Statistics

Table 2 lists a number of compile-time statistics collected for each program while applying PAD to our base cache. The first three columns describe each program. GLOBAL ARRAYS indicates the number of global (or globalized) arrays. % UNIF. REFS shows the percent of references in the program the compiler classifies as uniformly generated. The next four columns present statistics for intra-variable padding. ARRAYS SAFE is the number of arrays which can be safely padded if desired. We see that most references are uniformly generated, but the compiler finds many arrays in larger programs which cannot be safely padded. ARRAYS PADDED contains the number of arrays actually padded (none if blank). MAX # INCR is the maximum number of elements by which any array is padded. TOTAL # INCR is the total number of elements by which all arrays are padded. Though intra-variable padding is typically profitable when applied, PAD discovers opportunities for these transformations in only 6 out of our 36 benchmark programs.

The next column BYTES SKIPPED shows the total number of bytes used for inter-variable padding. We see inter-variable padding is needed more frequently, with the compiler applying padding to fifteen programs. The final column % SIZE INCR displays the percent increase in the total size of all global variables following padding. Results show only a small amount of padding is needed. The increase in memory size after both inter- and intra-variable padding is almost negligible, and is under 1% for all programs.

4.3 Impact on Cache Miss Rates

We now examine the effectiveness of padding heuristics in eliminating conflict misses in benchmark programs through cache simulations. We evaluate the impact of several factors, including transformations, analysis, associativity, and cache size. In the figures, the X-axis lists programs, the Y-axis presents cache miss rates or miss rate improvements, both in percentages. *Miss rate improvements* are calculated as the difference in the miss rate. Reducing the cache miss rate from 10% to 8% would yield an improvement of 2%, for instance. If the miss rate jumps from 10% to 12%, the degradation would yield an "improvement" of –2%.

Overall Impact Figure 8 presents cache miss rates for the original program and the version optimized by PAD. We see that conflict misses are eliminated for many programs, dropping the average miss rate from 16.8% to 7.9%. The average percentage reduction in miss rates for each program is 16%. Padding is more successful for kernels than entire applications.

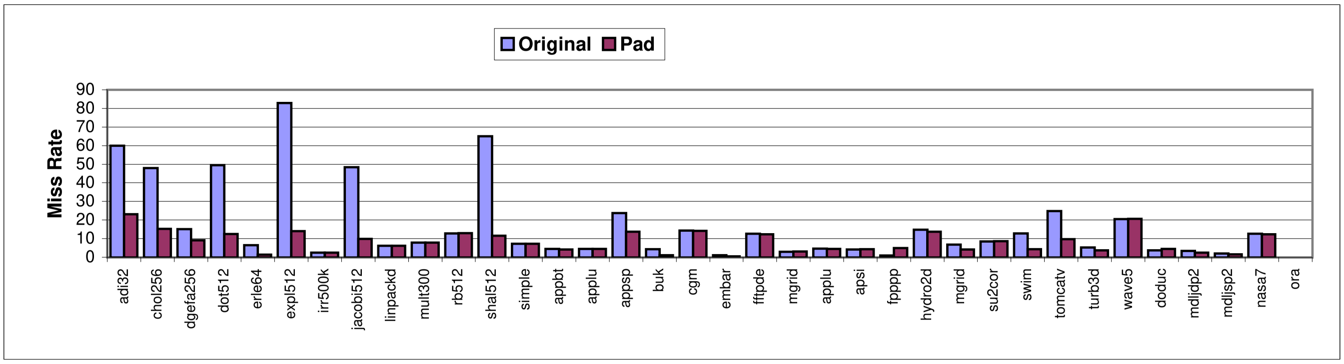

Figure 8: Impact of Padding Transformations (16K Direct-mapped Cache)

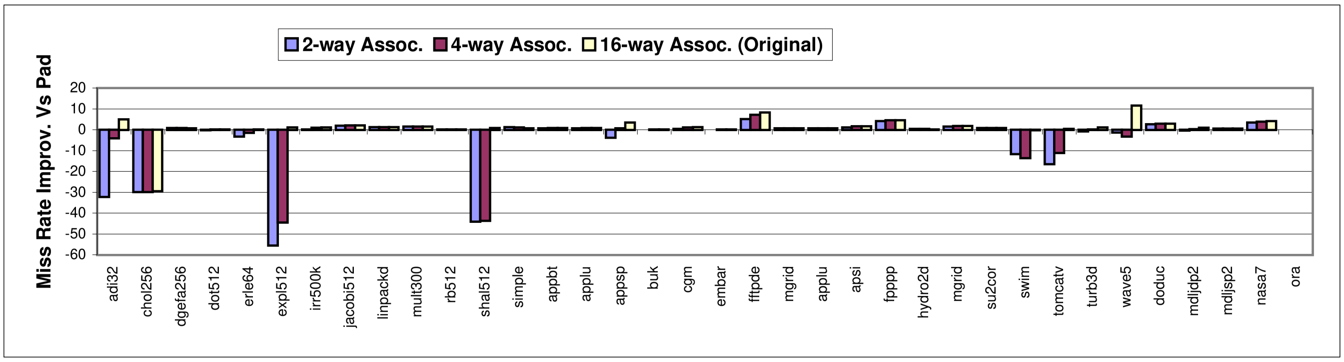

Figure 9: Comparing Padding to Higher Set Associativity (16K Cache)

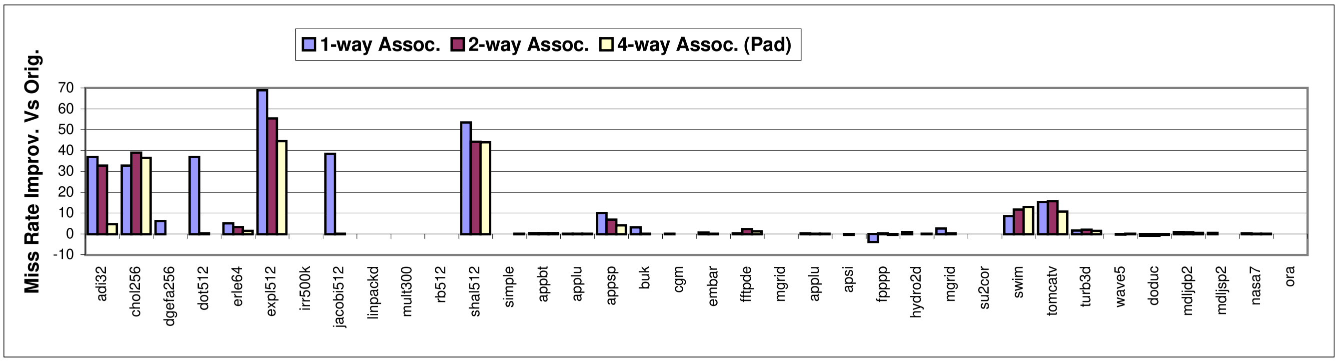

Figure 10: Impact of Associativity on Padding (16K Cache)

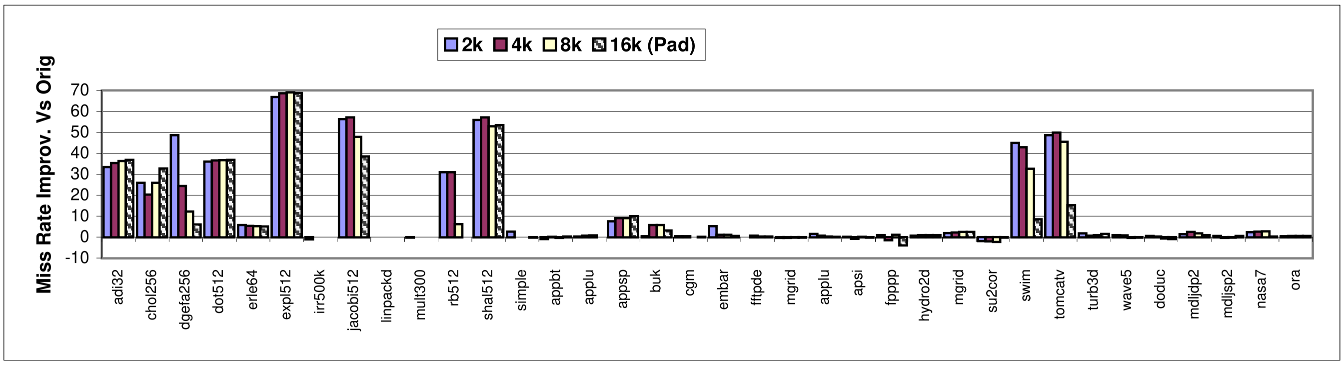

Figure 11: Impact of Cache Size on Padding (Direct-mapped Cache)

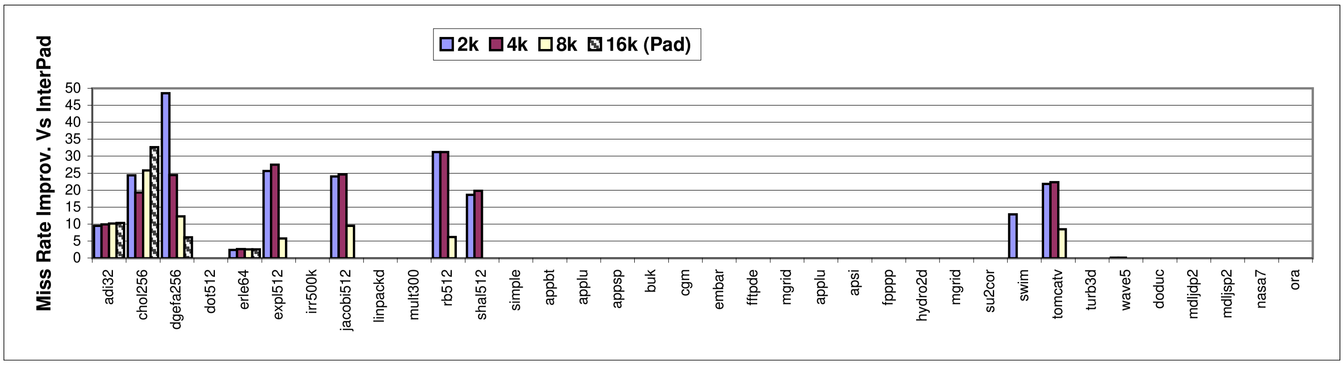

Figure 12: Impact of Cache Size on Intra-variable Padding (Direct-mapped Cache)

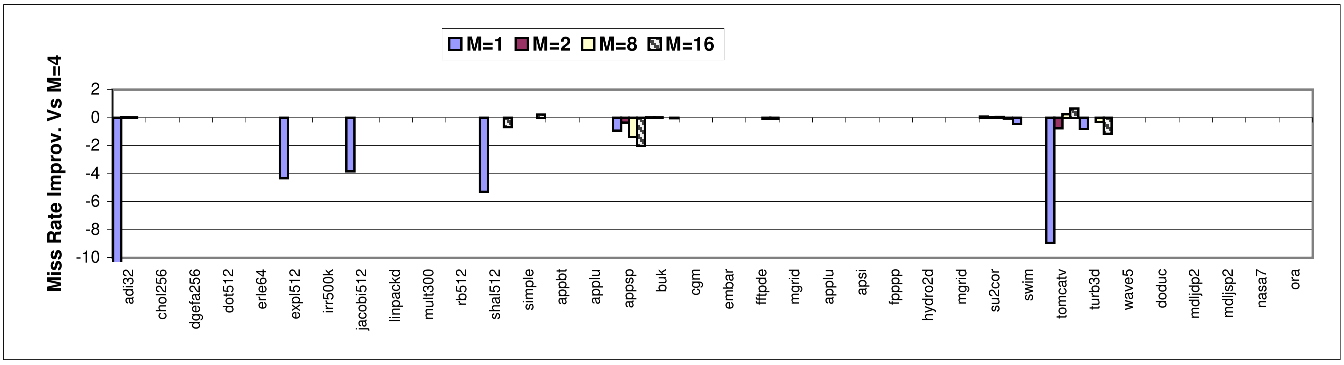

Figure 13: Impact of Minimum Inter-variable Separation for PadLite (16K Direct-mapped Cache)

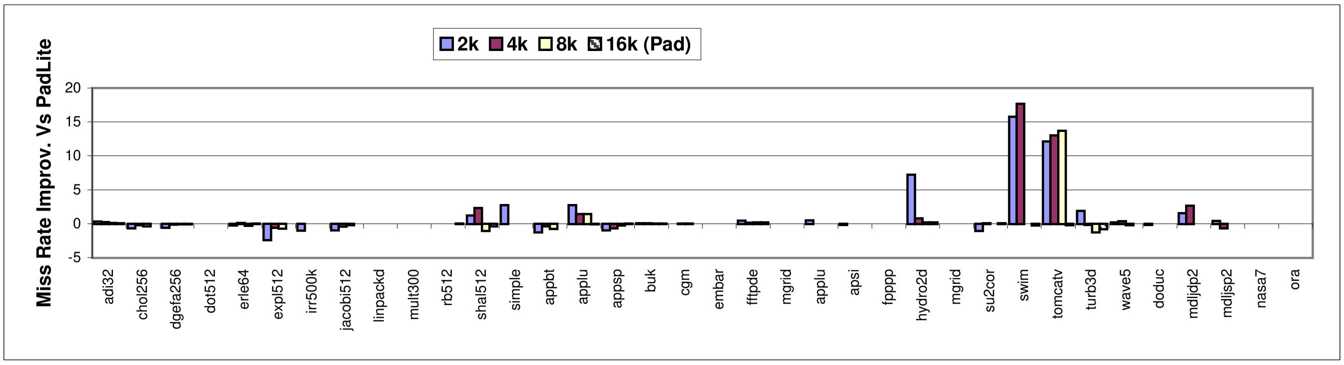

Figure 14: Impact of Precision of Analysis on Padding for Varying Cache Sizes (Direct-mapped Cache)

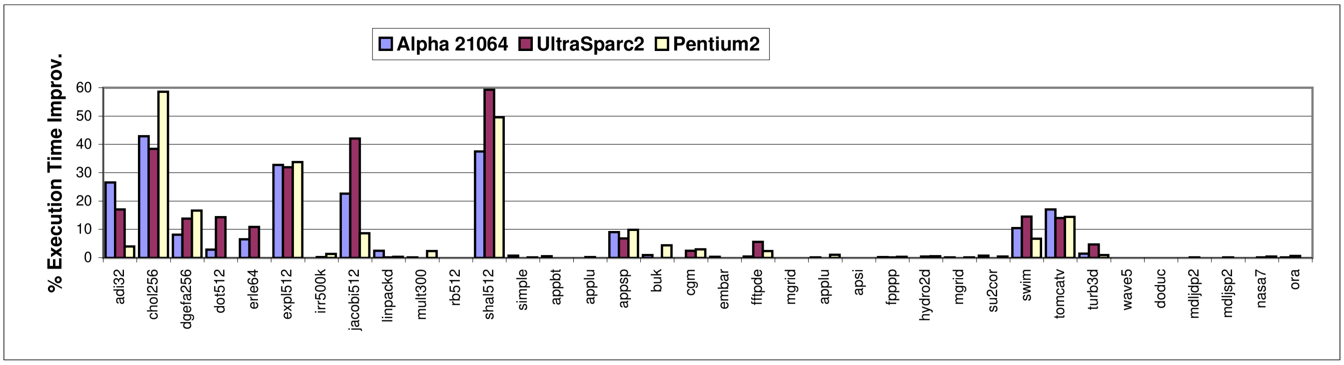

Figure 15: Impact of Padding on Program Execution Time

Associativity We examine the effect of higher associativity on padding in two ways. First, we compare miss rate improvements for padding with higher set associative caches in Figure 9. Differences are calculated between PAD on a direct-mapped cache and the original program on higher set-associative caches. Though PAD is not successful at eliminating conflicts in some cases (FFTPDE, FPPP, WAVE and NASA7), it is significantly more effective than 2-way and 4-way associative caches for many programs (ADI, CHOL, EXPL, SHAL, SWIM, TOMCATV). 16-way associative caches are required to achieve comparable improvements.

In Figure 10 we compare the impact of padding as set-associativity increases. Differences are calculated between PAD and the original program for 1-, 2-, and 4-way associative caches. Results show some programs (DGEFA, DOT, JACOBI) benefit from padding on direct-mapped caches but not for higher set-associative caches. Other programs experience decreasing benefits (ADI, EXPL, SHAL, APPSP, TOMCATV). Both sets of experiments show that the impact of padding decreases for higher set-associative caches, but can still be significant.

Effect of Cache Size on Padding Figure 11 demonstrates the impact of cache size on padding. Differences in miss rates between PAD and the original program are calculated for different cache sizes. In general, padding becomes more important for smaller caches, when the ratio of problem to cache size increases. For some programs (DGEFA, JACOBI, RB, SWIM, TOMCATV) padding becomes more effective as cache sizes decrease. In others (ADI, CHOL, DOT, EXPL, SHAL, APPSP) improvements remain constant.

Intra-variable Padding Figure 12 examines miss rates improvements due to intra-variable padding for different cache sizes. Differences in miss rates between PAD (inter- and intra-variable padding) and INTERPAD (inter-variable padding only) are calculated. Results show intra-variable padding is useful for only a few programs on a 16K cache, but has wider applicability as the cache size decreases. We always apply inter-variable padding after intra-variable padding, otherwise improvements may actually be due to changing array base addresses.

Minimum Separation Distance PADLITE separates variables by a minimal distance M (in terms of L_s) when applying padding. M must be large enough to avoid conflict misses in typical codes, yet small enough to avoid wasting memory. Figure 13 compares miss rates of PADLITE for different values of M, compared to $M = 4$. We see that $M = 1$ is insufficient for eliminating conflict misses in several programs. Other values of M yield miss rates similar to $M = 4$, except for APPSP and TURB3D. Over all programs, $M = 4$ is outperformed only by TOMCATV with $M = 8, 16$.

Precision of Analysis Figure 14 looks at miss rates improvements due to the precision of intra-variable padding for different cache sizes. The difference in miss rates for PAD and PADLITE are calculated. Results indicate the additional analysis of PAD is rarely useful on a 16K cache, but is more effective for smaller caches. Several programs (SHAL, SIMPLE, APPLU, HYDRO2D, SWM, TOMCATV, TURB3D, MDLJDP2) benefit on a 2K cache. PAD slightly increases miss rates for some codes (EXPL). Precise analysis thus becomes more important as opportunities for conflicts increase.

4.4 Impact on Execution Time

To determine the impact on actual program performance, original and padded versions of each program were timed on a DEC Alpha 21064, Sun UltraSparc2, and Intel Pentium2. Figure 15 presents the percentage improvement in execution time for PAD over the original program. Many kernels speed up significantly, but large improvements are only found for a few applications (APPSP, SWIM, TOMCATV). Results demonstrate that observed miss rate reductions translate into average execution time improvements of 6.0% for Alpha, 7.5% for UltraSparc2, and 5.9% for Pentium2.

4.5 Varying Problem Sizes

To compare the PADLITE and PAD heuristics more thoroughly, we examined cache simulations for four kernels as we varied the problem sizes from 250 to 520. Two stencil codes compute explicit hydrodynamics (EXPL) and shallow water modeling (SHAL), while two linear algebra codes compute Gaussian elimination with partial pivoting (DGEFA) and Cholesky factorization (CHOL). We measured miss rates for the original program (for both the base 16K direct-mapped cache, and a 16-way associative cache), as well as optimized versions of each program (PADLITE and PAD) on the base cache.

Optimizations Results are shown in Figure 16. The X-axis represents problem sizes, while the Y-axis presents the cache miss rate. We see each kernel experiences severe conflict misses on a direct-mapped cache for many problem sizes, particularly powers of two. Conflict misses are especially prevalent for CHOL. The 16-way associative cache can eliminates all cache conflicts, except for some problems sizes for CHOL. PADLITE is able to eliminate most conflict misses in EXPL, SHAL and DGEFA, but misses padding opportunities for many CHOL problems sizes. In comparison, PAD appears sufficiently powerful to consistently eliminate severe conflict misses for all four kernels. Compiler-directed padding can thus be beneficial in avoiding severe conflict misses. For CHOL padding sometimes even improves miss rates compared to the 16-way associative cache. PAD is shown to be more stable than PADLITE for the four kernels over the range of problem sizes tested.

Intra-variable Padding We also use varying problem sizes to compare the effectiveness of LINPAD1 and LINPAD2. To compare the two heuristics, we applied LINPAD1 or LINPAD2 followed by INTERPADLITE, subtracting the result from applying INTERPADLITE alone. Results are shown in Figure 17. The X-axis represents problem sizes, while the Y-axis presents the change in cache miss rate.

We see both LINPAD1 and LINPAD2 produce fairly random perturbations in cache miss rates for the stencil codes EXPL and SHAL, yielding both improvements and degradations. LINPAD2 applies padding frequently, leading to many large perturbations in cache miss rates. LINPAD1 applies padding occasionally, with only small changes in miss rates. These results justify our decision to use LINPAD1 in PADLITE. LINPAD2 is reserved for PAD, where it is applied only to linear algebra codes. PAD thus does not apply LINPAD2 to either EXPL or SHAL.

The benefits of LINPAD1 and LINPAD2 are visible in the miss rate improvements of linear algebra codes DGEFA and CHOL in Figure 17. For DGEFA, LINPAD1 is sufficient to eliminate severe conflicts. LINPAD2 does no harm but is not able to eliminate additional conflict misses. For CHOL LINPAD1 eliminates conflicts for several problem sizes, but LINPAD2 can find many additional problems sizes which benefit from intra-variable padding. Overall, the two heuristics appear quite useful for linear algebra codes, less so for stencil codes.

4.6 Discussion

In general, we found small inter-variable pads are sufficient to eliminate the worst of the severe conflict misses. Both PADLITE and PAD are successful for benchmark programs.

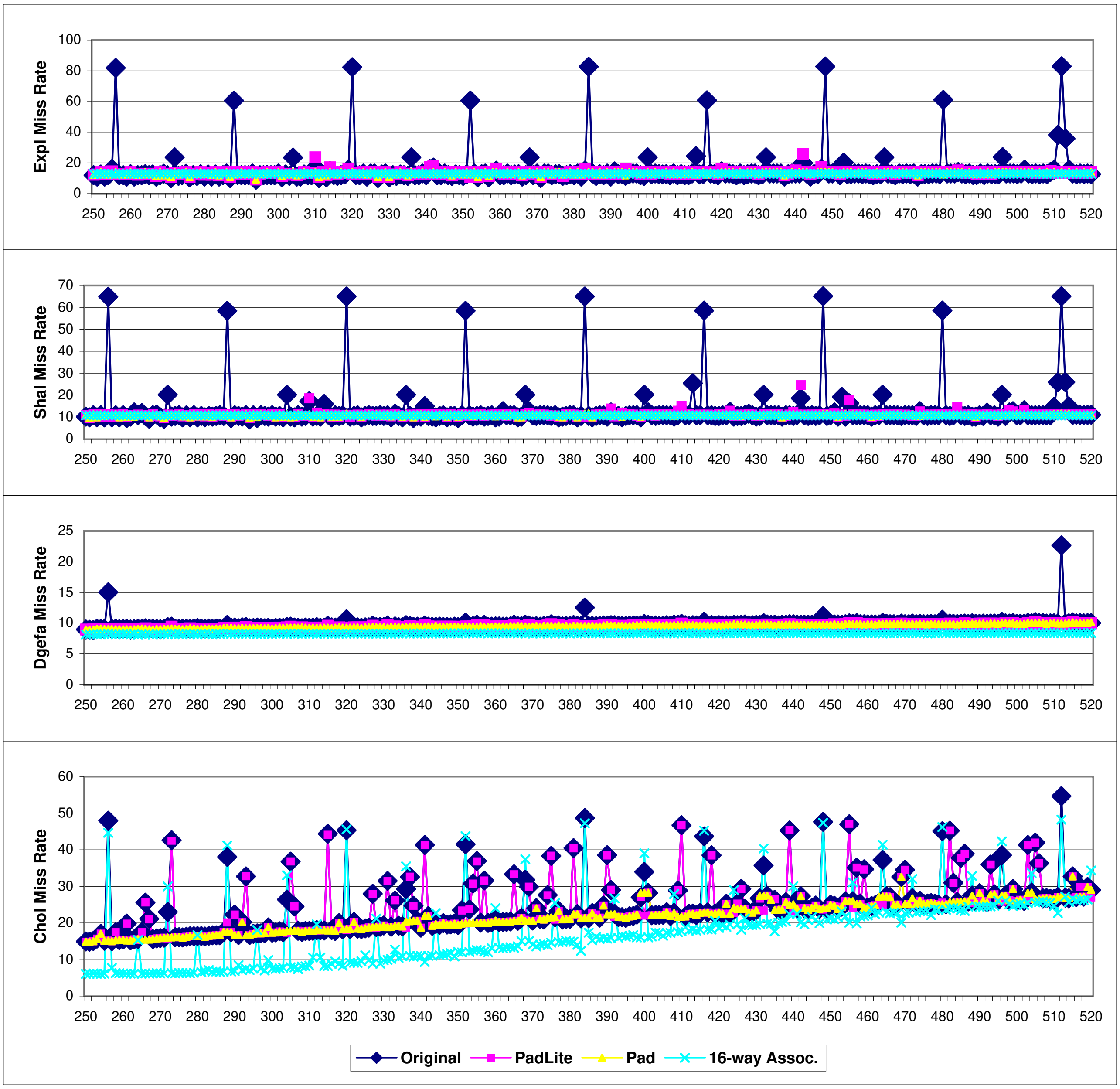

Figure 16: Impact of Optimizations on Cache Miss Rates for Varying Problem Sizes

However, Pad is more powerful, as demonstrated by results from smaller cache sizes and varying problem sizes. Many programs benefit from inter-variable padding, relatively few require intra-variable padding. Results indicate padding transformations become more important as the ratio of application data to cache size increases, so precise analysis may become more important for real-world programs with large data sets. Our results show not all programs were greatly improved, but the impact is significant for some programs. However, more powerful analysis is needed to prove the safety of padding for large programs.

5 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 [23]. Gannon *et al.* introduce the notion of uniformly generated references as a means of discovering group reuse between references to the same array [9]. Most compiler researchers have concentrated on computation-reordering transformations. Loop permutation and tiling are the primary optimization techniques [9, 17, 23], though loop fission (distribution) and loop fusion have also been found to be helpful [15, 17]. Coleman and McKinley show how to select tile sizes which avoid conflict misses using the Euclidean algorithm [7].

McKinley and Temam perform a study of loop-nest oriented cache behavior for scientific programs and conclude that conflict misses cause half of all cache misses and most intra-nest misses [18]. Researchers have examined methods to eliminate conflict misses using hardware [11, 13] or

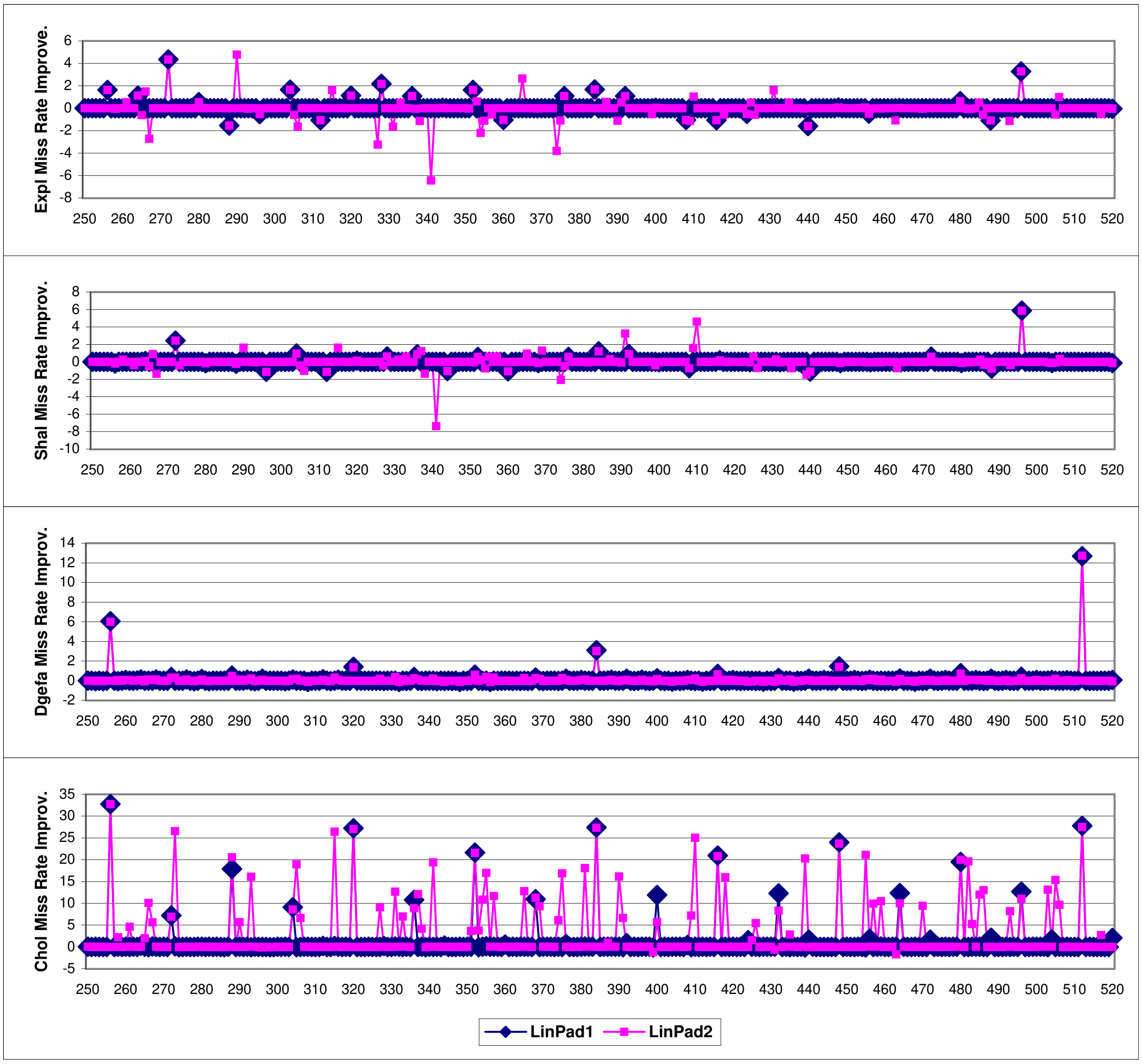

Figure 17: Impact of Intra-variable Padding on Cache Miss Rates for Varying Problem Sizes

operating systems support [2, 4]. These techniques are general and apply to a wide range of programs. However, for many scientific Fortran codes we can achieve similar or better performance using inexpensive data layout transformations. Many researchers have investigated the related problem of avoiding memory bank conflicts in vector systems [5], but do not discuss compiler techniques.

A number of researchers have examined data layout transformations, usually in the context of parallel programs. Jeremiassen and Eggers automatically eliminate false sharing in explicitly parallel programs by changing the placement of variables [12]. Amarasinghe *et al.* demonstrate the utility of data layout transformations for eliminating conflict misses in parallel applications [1]. They find it to be significant in eliminating adverse cache effects, though specialized optimizations are necessary to reduce computation overhead for modified array subscripts. Cierniak and Li examine combining array transpose and loop transformations to improve locality for parallel programs [6]. More recently, O'Boyle & Knijnenburg [19] and Kandemir *et al.* [14] investigate array transpose as a technique for improving data locality in uniprocessors. 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 [15]. McFarling shows compiler transformations of code sequences can help eliminate conflict misses in the instruction cache [16].

Many researchers have also examined the problem of deriving estimates of cache misses in order to help guide data locality optimizations [8, 9, 23]. These models typically can predict only capacity misses because they assume a fully-associative cache. Temam *et al.* establish that conflict misses can significantly degrade performance and present an analytical method for detecting and counting the number of cache misses due to conflicts [20]. Ghosh *et al.* calculate conflict misses more 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 [10]. They demonstrate how to use cache miss equations to select array paddings to eliminate conflict misses and block sizes for tiling.

Because of the complexity and expense of their more precise algorithms, both Temam *et al.* and Ghosh *et al.* have only evaluated their algorithms on a few kernels. In comparison, we use a simplified version of cache miss equations to detect when large numbers of conflict misses will occur, then apply padding until severe conflicts are mostly eliminated. Because we avoid calculating the number of conflict misses directly, we can inexpensively guide padding transformations for much larger programs than before.

6 Conclusions

In this paper, we develop a number of techniques for eliminating cache conflicts misses in scientific programs. We develop and evaluate heuristics for guiding two data-layout transformations, inter- and intra-variable padding. We demonstrate the practicality of our heuristics by applying them to a large collection of kernels and benchmark programs over a range of cache and problem sizes. Our heuristics are limited to regular computation patterns; more research is needed to extend them to a wider range of programs. However, preliminary experimental results for scientific Fortran codes are encouraging. Our prototype compiler achieves respectable improvements on average and significant improvements for some programs. Our results indicate data layout transformations should become a standard optimization for advanced compilers. As processor speeds continue to increase relative to memory latencies, locality optimizations should become even more important for future processors.

References

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

[2] 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] 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.

[4] 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.

[5] D. Calahan. Performance evaluation of static and dynamic memory systems on the Cray-2. In *Proceedings of the Second International Conference on Supercomputing*, St. Malo, France, July 1988.

[6] 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.

[7] 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.

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

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

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] S. McFarling. Program optimization for instruction caches. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pages 257–270, Boston, MA, April 1989.

[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] 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.

[21] 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.

[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.