Locality Optimizations for Multi-Level Caches

Gabriel Rivera, Chau-Wen Tseng

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

Abstract

Compiler transformations can significantly improve data locality of scientific programs. In this paper, we examine the impact of multi-level caches on data locality optimizations. We find nearly all the benefits can be achieved by simply targeting the L1 (primary) cache. Most locality transformations are unaffected because they improve reuse for all levels of the cache; however, some optimizations can be enhanced. Inter-variable padding can take advantage of modular arithmetic to eliminate conflict misses and preserve group reuse on multiple cache levels. Loop fusion can balance increasing group reuse for the L2 (secondary) cache at the expense of losing group reuse at the smaller L1 cache. Tiling for the L1 cache also exploits locality available in the L2 cache. Experiments show enhanced algorithms are able to reduce cache misses, but performance improvements are rarely significant. Our results indicate existing compiler optimizations are usually sufficient to achieve good performance for multi-level caches.

1 Introduction

Because of the increasing disparity between memory and processor speeds, effectively exploiting caches is widely regarded as the key to achieving good performance on modern microprocessors. As the gap between processors and memory grows, architects are beginning to rely on several levels of cache in order to hide memory latencies. Two level caches are now common, while processors such as the DEC Alpha 21164 have three levels of cache.

Compiler transformations for improving data locality can be useful in hiding the complexities of the memory hierarchy from scientists and engineers. Compilers may either rearrange the computation order through loop transformations (e.g., loop permutation, fusion, tiling), or change the layout of data through data transformations (e.g., padding, transpose). Such transformations are usually guided by a simple cache model to evaluate different optimization choices. In almost all cases, current state of the art models a single level of cache.

In this paper, we examine the impact of multi-level caches on data locality optimizations. We wish to 1) discover whether compiler optimizations need to consider multiple levels of cache in order to maximize program performance, and 2) quantify the improvements possible by targeting multi-level caches.

We find that many compiler transformations improve locality for all levels of cache simultaneously, without any need to explicitly consider multiple cache levels. In other cases, targeting just the smallest cache generally yields most of the benefits of explicit optimizations for the larger cache(s). We show several transformations which can be enhanced for multi-level caches. In these cases we propose new heuristics and evaluate their impact. The contributions for this paper are:

- Examining when data locality optimizations will benefit from targeting multi-level caches.
- Developing new heuristics for inter-variable padding, loop fusion, and tiling for multi-level caches.
- Experimentally evaluating the impact of multi-level cache optimizations. Results show while miss rates may be reduced, performance is not improved except in rare cases.

For simplicity, we usually consider a multi-level cache of two levels, with a small level-one primary cache (L1) and a large level-two secondary cache (L2). In modern microprocessors, the L1 cache is typically 8K or 16K, while the L2 cache is 96K to 4M. We also assume both the L1 and L2 caches are direct-mapped. Optimizations which avoid conflict misses on a direct-mapped cache certainly avoid conflicts in k-way associative caches. Our experience indicates that simply treating k-way associative caches as direct-mapped for locality optimizations achieves nearly all the benefits of explicitly considering higher associativity.

In the rest of the paper, we begin by considering why many locality optimizations do not need to consider multiple levels of cache, then focus on individual transforma-

```
// Original              // Loop Permutation      // Array Transpose
real A(N,M), B(N)        real A(N,M), B(N)        real A(M,N), B(N)
do j = 1,N               do i = 1,M               do j = 1,N
 do i = 1,M               do j = 1,N               do i = 1,M
  B(j) = A(j,i)            B(j) = A(j,i)            B(j) = A(i,j)
```

Figure 1 Loop Nest and Data Layout Transformations

tions which may benefit from targeting multi-level caches. We describe how to extend inter-variable padding, loop fusion, and tiling for multi-level caches. We experimentally evaluate our optimizations on a selection of representative programs through cache simulations and actual program execution and discuss our results. We describe related work and then conclude.

2 Locality Optimizations

When we examine compiler transformations for improving data locality, we find many loop nest and data layout transformations do not need to explicitly target multi-level caches. To see why this is the case, we briefly review different forms of data locality. As pointed out by Wolf and Lam [30], reuse can be either *temporal* (multiple accesses to same memory location) or *spatial* (multiple accesses to nearby memory locations). If reuse is between multiple references to the same variable, it is considered *group* reuse, otherwise it is *self* reuse.

Consider the example Fortran code in Figure 1. In the code there is temporal reuse of `B(j)` on the i loop because all iterations of i will access the same memory location. There is spatial reuse of `B(j)` on the j loop because iterations of j will access consecutive locations in memory. Similarly, there is spatial reuse of `A(j,i)` on the j loop, because on iterations of j the reference will access consecutive memory locations in A (arrays are column-major in Fortran).

In the original version of the program, spatial reuse of `A` is unlikely to be exploited for large values of M, since iterations of loop i access many different cache lines. To improve data locality, we can apply loop nest and/or data layout transformations.

2.1 Loop nest transformations

Loop nest transformations can change both the order loop iterations are executed, and structure of the loop nest. The most basic loop nest transformation is loop permutation, which changes the order of loops in a loop nest. Consider applying loop permutation to the code in Figure 1. Now that the j loop is innermost, both the temporal reuse of B and spatial reuse of A occur much closer in time, on consecutive loop iterations rather than across M iterations of the i loop. As a result, for large values of M, loop permutation significantly increases the probability the cache line being accessed is still in the cache, improving performance.

Notice that bringing reuse closer together in time is almost always desirable, regardless of cache size or level. The only role cache size plays is whether the cache is large enough to keep data in the cache across M iterations of the i loop in the original program. If not, then miss rates should drop for that level of cache. If the change is significant, it will produce a performance improvement. Loop permutation in this example thus benefits all levels of cache simultaneously. For small values of N, M, only upper levels of cache will benefit. For large enough values of N, M, all levels of cache will benefit. The compiler does not have to explicitly target multiple levels of cache.

In addition to loop permutation, other unimodular loop nest transformations such as *loop skewing*, *loop reversal*, and their combinations [30, 31] do not need to target multi-level caches, for reasons just described. One issue which may arise is if tradeoffs must be made between spatial and temporal locality in choosing a profitable loop permutation. In such cases the cache line size may affect the impact of spatial locality. So if cache line sizes differ significantly for different levels of cache, loop nest transformations may need to be aware of multi-level caches. We have not found any such cases in practice.

2.2 Data layout transformations

Another class of location optimizations modify the data layout, rather than the loop iterations. Data layout transformations are designed to improve only spatial locality. An example of *array transpose* [5, 13], a simple data layout transformation, is shown in Figure 1. Array A is transposed, changing the reference `A(j,i)` to `A(i,j)`. Consecutive accesses are now to adjacent memory locations, increasing the the chance they will access the same cache line.

As with loop permutation, data layout transformations benefit multiple levels of cache simultaneously, since bringing references close in memory improves spatial locality at all cache levels. Cache line size is important in determining whether transformations will actually reduce misses; improvements will result only if accesses are close enough to end up on the same cache line. If so, the local-

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

do j = 2,N-1    // loop nest 1
 do i = 1,N
  = A(i,j) + A(i,j+1)
  = B(i,j) + B(i,j+1)
  = C(i,j) + C(i,j+1)

do j = 2,N-1    // loop nest 2
 do i = 1,N
  = B(i,j-1) + B(i,j) + B(i,j+1)
  = C(i,j)
```

Figure 2 Example program

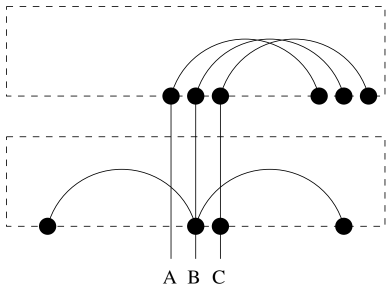


Figure 3 PAD layout for example

ity optimization can improve cache line utilization, reduce working set size, and exploit hardware prefetching.

In addition to array transpose, other unimodular [12] and nonlinear [4] array layout transformations also do not need to target multiple levels of cache, for the same reasons as just discussed. One issue which may arise is if cache line sizes are significantly different in size. If data layout transformations incur overhead, then how successful the transformation is in putting data accesses onto the same cache line becomes a factor in deciding whether to perform the transformation. In this case the cache line size (and miss penalty) for each cache level must be considered.

We have seen how many loop nest and data layout transformations are largely insensitive to multi-level caches. However, several optimizations can benefit from targeting multiple levels of cache. We examine them in the following sections.

3 Inter-variable Padding

Inter-variable padding is a data transformation useful for eliminating conflict misses [1, 21, 22]. Existing methods for inter-variable padding generally require knowledge of cache size and line size. To see how padding variables eliminates conflict misses at a single cache level, consider the program in Figure 2. 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 map to the same cache line in the first loop nest, eliminating reuse. In this case *severe* or *ping-pong* conflict misses result, since misses can occur on every iteration. To avoid 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.

There is also *group reuse* of columns of `B` carried on the outer `j` loop in both nests, 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. As we review in the next section, inter-variable padding is also useful for preserving such group reuse.

3.1 Avoiding Severe Conflicts

3.1.1 Current Methods

PAD is a simple compiler technique for eliminating severe conflict misses [21]. It analyzes array subscripts in loop nests to compute a memory access pattern for each array variable. It then iteratively increments each variable base addresses until no conflicts result with other variables analyzed. When considering a base address for a variable A, if PAD finds a loop in which a reference to A maps to a location on the cache within one cache line of a reference to a different variable, PAD will increment the address until conflicts are eliminated. In practice, PAD requires only a few caches lines of padding per variable [21].

Figure 3 illustrates a possible layout achieved by PAD when transforming the layout of Figure 2 to avoid severe conflicts on the L1 cache. Each box corresponds to the L1 cache during a given loop nest, with the width representing the cache size. In this case, the cache size is slightly more than double the common column 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 `A(i,j)` in the first loop is `A(i,j+1)`. The two dots are thus apart by a distance of `N`, the column size. Since all references are *uniformly generated*, these relative positions do not change over loop iterations.

Layout diagrams such as Figure 3 appear in several places in this paper. These 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 would be illustrated by superimposing dots. 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. 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 the L1 cache size, we find all base addresses in the original sample program coincide on the cache, causing severe conflicts between references to different arrays. In Figure 3, we see PAD eliminates severe conflicts by inserting small pads between successive variables, so no dots overlap. However, since dots appear between the endpoints of 4 out of 5 arcs, group reuse is not fully exploited. For instance, the reuse of `B(i,j+1)` by `B(i,j)` in the second loop is prevented by `C(i,j)`.

3.1.2 Multi-level Methods

The PAD algorithm generalizes easily to multiple cache levels. Base addresses are tested for conflicts with respect to all cache levels instead of just one cache. An even simpler method follows from the fact that the cache size at a given level evenly divides that of lower levels. Base addresses are padded when conflicts result with respect to a single cache configuration. This configuration consists of the L1 cache size, S_1, and the largest cache line size found at any level, L_{max}. Note that S_1 is the smallest cache size at any level. Thus, when each cache level shares a common line size, this configuration is the same as the L1 cache. Otherwise, this precise configuration does not actually exist in the memory hierarchy. We call this MULTILVLPAD since it generalizes pad for multiple levels of cache.

The validity of this method follows from modular arithmetic. If two references maintain a distance of at least L_{max} on a cache of size S_1, then the distance must be equal or greater on a cache of size kS_1 (larger by an integer factor k). Spacing references by at least L_{max} ensures severe conflicts are avoided at each cache level regardless of line size.

3.2 Preserving Group Reuse

3.2.1 Current Methods

Often the L1 cache contains sufficient space with which to exploit some or all group temporal reuse across outer loop iterations. We previously introduced GROUPPAD which inserts larger pads than PAD to obtain a layout both preserving group reuse on the L1 cache and avoiding severe conflict misses [22]. Figure 4 gives the result of applying GROUPPAD to the example program using a diagram similar to Figure 3. By sufficiently separating `B` from `A` and `C` on the cache, all group reuse between `B` references is preserved. Though `A` and `C` references fail to exploit group reuse, it is apparent that the L1 cache lacks the capacity to preserve all group reuse in the first loop (as this would require a cache size three times the column size.) It is thus unavoidable that two out of three arcs must overlap.

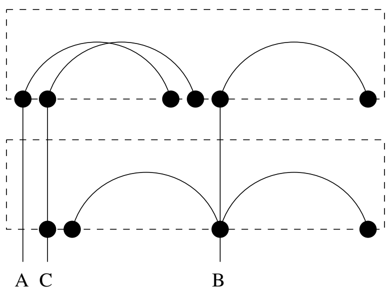


Figure 4 GROUPPAD layout for example on L1 cache

GROUPPAD obtains such a layout by considering for each variable a limited number of positions relative to other variables. The number of references successfully exploiting group reuse at the L1 cache is counted for each position. GROUPPAD then selects the position maximizing this value.

3.2.2 Multi-level Methods

Once we consider a secondary cache, another goal emerges: to preserve on the L2 cache the group reuse which remains unexploited on the L1 cache following GROUPPAD. In order to preserve the GROUPPAD layout on the L1 cache, we restrict pad sizes to multiples of S_1, the size of the L1 cache. If we pad a variable B by such an amount, the relative distances between B references and references to other variables may change on the L2 cache but not on the L1 cache. By using only S_1 sized pads, we can then reapply GROUPPAD to optimize group reuse for the L2 cache. We can thus GROUPPAD in such a manner that it begins targeting the L1 cache as already described, and then in later phases recursively applies GROUPPAD to exploit group reuse for lower levels of cache, using pads which are multiples of the previous cache size to preserve group reuse at higher levels of cache.

For large L2 caches, a simpler approach may sometimes suffice. If array column sizes are a small fraction of the L2 cache size, merely spacing variables as far apart as possible on the L2 cache can preserve all group reuse at this cache level. This is illustrated in Figure 5. Boxes

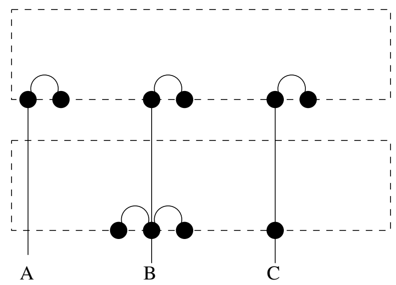


Figure 5 L2MAXPAD layout for example on L2 cache

now represent the much larger L2 cache. We see that all group reuse is exploited on this cache.

To preserve the L1 cache layout computed by GROUPPAD while separating variables in this manner, we also round pads to the nearest S_1 multiple after determining the approximate position for a variable on the L2 cache. In this way we can maintain the layout of Figure 5 on the L2 cache while simultaneously maintaining the layout of Figure 4 on the L1 cache. This will allow `A` and `C` references in the first loop to exploit group reuse on the L2 cache where they do not on the L1 cache. We call this method L2MAXPAD, since it extends the our MAXPAD algorithm [22].

3.3 Summary

Transformations for the L2 cache combine easily with PAD and GROUPPAD. To eliminate severe conflict misses at both cache levels, MULTILVLPAD need only use the largest line size instead of the L1 cache line size. To preserve all possible group reuse at both cache levels, we run GROUPPAD for the L1 cache and then apply further padding in multiples of the L1 cache size to enable additional L2 group reuse. These techniques easily generalize to three or more cache levels.

4 Loop Fusion

Loop fusion is a transformation where adjacent loops are fused into a single loop containing both loop bodies. It can be used to improve locality directly by bringing together memory references [14, 18, 25], or to enable additional locality optimizations such as loop permutation [19] and array contraction [9].

We observe improvements in temporal locality after fusing the loop nests of Figure 2 at the innermost level, obtaining the nest shown in Figure 6. Assuming array sizes exceed the L2 cache size, in the original loop nest reference `B(i,j+1)` would miss both cache levels in both loops. Fusing the loops ensures that only one memory access is needed; the second reference to `B(i,j+1)` may be found in cache or register. Fused loops may therefore exhibit improved temporal locality.

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

Figure 6 Example program after fusion

Loop fusion can improve data locality, but it also increases the chance of severe conflicts. We find applying inter-variable padding using the PAD algorithm after loop fusion is important. Fortunately, it can eliminate severe conflicts on all levels of cache fairly easily.

Another disadvantage to loop fusion is that the increased amount of data accessed per loop iteration can force a loss of group temporal reuse on smaller caches. Consider Figure 7, which illustrates the layout of the fused nest after GROUPPAD. This figure consists of only one box, since the two loops have been fused into one. Note that unlike in earlier diagrams, dots may represent two identical references, due to fusion. We see that on the L1 cache, group reuse is exploited only for one reference, `B(i,j-1)`. A L1 cache size over four times the column size would be required to exploit all group reuse.

Since three references exhibit group reuse in Figure 4 we find that loop fusion has decreased the overall amount of group reuse exploited on the L1 cache. To get a precise accounting of the cache effects of fusing the two loops, we can count the total number of references which due to cache faults access either the L2 cache or main memory. We compute these totals for original and fused versions. We assume L2MAXPAD is applied following GROUPPAD so that group reuse is exploited on the L2 cache whenever it is not on the L1 cache. We also assume no reuse between nests due to capacity constraints.

In Figure 4, we see that references `A(i,j+1)`, `B(i,j+1)`, and `C(i,j+1)` in the first loop must access main memory, as do `B(i,j+1)` and `C(i,j)` in the second, totaling five memory references. Since `A(i,j)` and `C(i,j)` in the first loop do not exploit group reuse on the L1 cache, they must access the L2 cache. The remaining references (all to `B`) successfully exploit group reuse on the L1 cache. In total, two references access the L2 cache. Of course due to self-spatial reuse, these cache faults occur only whenever a reference accesses a new cache line. In the fused loop of Figure 7, three references, `A(i,j+1)`, `B(i,j+1)`, and `C(i,j+1)` must access main memory, an improvement from Figure 4. However,

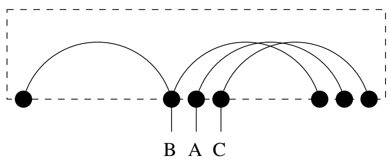


Figure 7 GROUPPAD layout for example on L1 cache after fusion

```
do KK=1,N,W          // W = tile width
 do II=1,N,H         // H = tile height
  do J=1,N
   do K=KK,min(KK+W-1,N)
    do I=II,min(II+H-1,N)
     C(I,J) = C(I,J) + A(I,K)*B(K,J)
```

Figure 8 Tiled matrix multiplication

three references, `A(i,j)`, `B(i,j)`, and `C(i,j)` will access the L2 cache. Note that wherever there are two identical references, only the first may cause a cache fault; the second will access the L1 cache or a register. Fusion has therefore saved two memory accesses but cost one L2 cache access.

We find therefore that loop fusion may involve a trade-off between L1 and L2 cache performance. Fortunately, the compiler can predict group reuse exploited at each cache level before and after loop fusion. Deciding whether fusion is profitable requires comparing the sum of reuse at each cache level, scaled by the cost of cache misses at that level. Since the cost of L2 misses are typically much higher than L1 misses, fusion will generally be profitable if it enables the compiler to exploit more L2 reuse.

5 Tiling

Tiling or *blocking* is a loop transformation which combines strip-mining with loop permutation to form small tiles of loop iterations which are executed together to exploit data locality [2, 11, 33]. Effectively utilizing the cache also requires avoiding conflict misses from tile self-interference using techniques such as tile size selection, intra-variable padding, and copying tiles to contiguous buffers [6, 3, 23].

Figure 8 illustrates a tiled version of matrix multiplication of `NxN` arrays in which reference `A(I,K)` accesses a `W` by `H` tile on each `J` loop iteration. When this tile fits in cache with no self-interference, data for array `A` is brought into cache just once, exploiting much temporal reuse. Data for the other arrays are brought in multiple times, causing a number of cache misses proportional to $\frac{1}{H} + \frac{1}{W}$ [6, 23]. When targeting a single level of cache, selecting the largest tile size which fits in cache will thus yields the fewest number of cache misses.

Tiling for multi-level caches is more complex. For each level of cache, selecting a tile larger than the cache will cause `A` to overflow, requiring it be read in `N` times. Smaller tiles, however, will cause more misses for arrays `B` and `C`. What tile size should be selected thus depends on the relative cost of misses at different levels of cache.

We believe most of the benefits of tiling may be obtained by simply choosing tile sizes targeting the L1 cache. First, from modular arithmetic we can show tiles with no L1 self-interference conflict misses will also have no L2 conflicts. Tiling for the L1 cache thus maximizes L1 reuse and also captures L2 reuse.

In comparison, choosing L2-sized tiles can reduce misses for `B` and `C` in the L2 cache but loses almost all L1 temporal reuse for `A`. Benefits are moderate since reductions in miss rates for the L2 cache from larger tiles grows slowly. For instance, quadrupling the size of a tile only reduces misses by 50% (to $\frac{1}{2H} + \frac{1}{2W}$) while the amount of reuse lost for L1 is proportional to N. As a result, tiling for the highest level of cache is likely to be more profitable unless the cost of L2 misses is much greater than for L1 misses. To choose the desirable tile size, the compiler can compare estimated cache misses at each cache level, scaled by their costs.

We note two exceptions to simply tiling for L1 caches. Song and Li recently extended tiling techniques to handle multiple loop nests enclosed in a single time-step loop, allowing tiles be overlapped from different time steps [26]. Because the data that must be held in cache spans many loop nests, tiles are typically larger than the L1 cache. As a result the tiling algorithm targets the L2 cache, completely bypassing the L1 cache.

Whaley and Dongarra use in their ATLAS library a refinement to our tiling algorithm [28]. They also pick L1-sized tiles, but impose an additional level of tiling so that tiles are traversed in panels designed to improve L2 performance. This additional tiling only yields improvement when the L2 cache is small relative to the L1 cache, otherwise the L2 cache can hold sufficient L1-sized tiles so that extra tiling is not necessary.

6 Experimental Evaluation

6.1 Evaluation Framework

We experimentally evaluated multi-level locality transformations for uniprocessors using both cache simulations and timings. Cache simulations were made for a 16K direct-mapped L1 cache with 32 byte cache lines and a 512K direct-mapped L2 cache with 64 byte lines. Miss rates for both the L1 and L2 cache are reported as the number of cache misses for that level, relative to the total

Program	Description	Lines
	KERNELS	
ADI32	2D ADI Integration Fragment (Liv8)	63
DOT256	Vector Dot Product (Liv3)	32
ERLE64	3D Tridiagonal Solver	612
EXPL512	2D Explicit Hydrodynamics (Liv18)	59
IRR500K	Relaxation over Irregular Mesh	196
JACOBI512	2D Jacobi with Convergence Test	52
LINPACKD	Gaussian Elimination w/Pivoting	795
SHAL512	Shallow Water Model	227
	NAS BENCHMARKS	
APPBT	Block-Tridiagonal PDE Solver	4441
APPLU	Parabolic/Elliptic PDE Solver	3417
APPSP	Scalar-Pentadiagonal PDE Solver	3991
BUK	Integer Bucket Sort	305
CGM	Sparse Conjugate Gradient	855
EMBAR	Monte Carlo	265
FFTPDE	3D Fast Fourier Transform	773
MGRID	Multigrid Solver	680
	SPEC95 BENCHMARKS	
APSI	Pseudospectral Air Pollution	7361
FPPPP	2 Electron Integral Derivative	2784
HYDRO2D	Navier-Stokes	4292
SU2COR	Quantum Physics	2332
SWIM	Vector Shallow Water Model	429
TOMCATV	Mesh Generation	190
TURB3D	Isotropic Turbulence	2100
WAVE5	Maxwell's Equations	7764

Table 1 Test programs for experiments

number of memory references (i.e., L2 misses are normalized to L1 misses). Timings were made on a Sun UltraSparc I, which has the same cache configuration as in our simulations. Transformations were applied to a number of scientific kernels and applications from NAS and SPEC95 benchmarks, shown in Table 1.

All data transformations were implemented as passes in the Stanford SUIF compiler [29]. Before these passes, some transformations are performed to give the compiler control over variable base addresses. First, local array and structure variables are promoted into the global scope. Formal parameters to functions cannot and do not need to be promoted, since they represent variables declared elsewhere. Array and structure global variables are then made into fields of a large structured variable, resulting in a single global variable containing all of the variables to be optimized. Optimizing passes may now modify the base addresses of variables by reordering fields in the structure and inserting pad variables. Also, intra-variable (array column) padding is first performed in ADI32 and ERLE64 to avoid severe conflicts between references to the same variable as described in [21]. Loop fusion and tiling were applied by hand, followed by padding. Tile dimensions and pad sizes were chosen automatically.

6.2 Padding To Avoid Severe Conflicts

To examine the effectiveness of data transformations for eliminating severe conflicts on multi-level caches, we transformed several programs. Performance was measured for three versions: original, optimized by PAD for only the L1 cache, and optimized by MULTILVLPAD for both caches (L1&L2). Figure 9 gives the simulated cache miss rates and execution time improvements for these programs.

The top two graphs present L1 and L2 cache miss rates. From the *L2 cache w/ L1 Opt* miss rates we note that PAD, though unaware of the L2 cache, obtains a L2 miss rate reduction similar in magnitude to the L1 reduction. To account for this L2 reduction, we may note that by improving spatial locality on the L1 cache, we avoid accesses to the L2 cache which might otherwise conflict.

We therefore find from *L2 cache w/ L1&L2 Opt* that MULTILVLPAD performs only slightly better on the L2 cache than PAD, mostly in the case of EXPL. By noting the similarity between values in *L1 cache w/ L1 Opt* and *L1 cache w/ L1&L2 Opt*, we also find that generalizing PAD for two levels of cache does not have an adverse effect on L1 miss rates.

The final graph in Figure 9 gives the UltraSparc execution time improvement relative to the original program for PAD and MULTILVLPAD versions. Timings were made for programs showing large miss rate changes in cache simulations. Comparing the two versions we find that multi-level optimizations for eliminating severe conflicts have only a minor effect on performance on this architecture, even slightly degrading performance in cases such as ERLE64. Reductions in L2 cache miss rates thus do not translate into performance improvements[1].

6.3 Padding To Preserve Group Reuse

6.3.1 Test Programs

We evaluated multi-level data transformations for preserving group reuse using five programs with numerous opportunities for improving group reuse. The results appear in Figure 10. The *L1 Opt* versions were transformed with GROUPPAD alone while the *L1&L2 Opt* versions were transformed by both GROUPPAD and L2MAXPAD. The first graph gives cache miss rates at both cache levels for the three versions. Again, we find that optimizing for the L2 cache in addition to the L1 cache is needed in few programs; only EXPL benefited on the L2 cache. L1 optimizations again account for most of the improvement in L2 cache miss rates. As we also see in Figure 9, optimizing for the L2 cache does not adversely affect L1 miss rates. Thus no inherent tradeoff exists between data transformations for the L1 cache and L2 cache.

The second graph demonstrates a very small improvement in EXPL execution time but improvements also in programs whose cache performance does not benefit from L2MAXPAD, the L2 transformation. Small degradations also occur in SWIM and TOMCATV, again suggesting that L2 optimizations have a small impact on this architecture.

6.3.2 Varying Problem Size

Prior work has shown that data transformations can be particularly useful for pathological problem sizes which might not arise in a limited set of test programs [22]. To reveal such opportunities for L2 data transformations and to investigate the robustness of these transformation, we varied the problem size of two programs, EXPL and SHAL, from 250 to 520 and simulated cache miss rates on both caches for *L1 Opt* and *L1&L2 Opt* versions. Results appear in figure 11, where the X-axis represents problem size and the Y-axis gives the cache miss rate.

We see that while both versions have similar L1 miss rates, *L1 Opt* (GROUPPAD alone) experiences clusters of problem sizes where L2 miss rates increase by up to 5%. The *L1&L2 Opt* versions avoid these increases. These results indicate that the clusters correspond to problem sizes in which overlapping array columns of different variables prevent group-temporal reuse or self-spatial reuse on the L2 cache. While for most problem sizes the distance between references is large on the L2 cache, references occasionally converge on one another as the problem size is varied. L2MAXPAD prevents this by separating variables on the L2 cache.

A prominent feature of these graphs is the invariant L2 miss rate of *L1&L2 Opt*, in contrast to the occasionally increasing L1 miss rates for both versions. This is attributed to the relative capacities of the two caches. The L1 cache, which can hold only 3 to 8 columns, depending on problem size, increasingly lacks the capacity to preserve group reuse as the problem size increases. All group reuse is exploited on the much larger L2 cache following L2MAXPAD.

6.4 Loop Fusion

To explore the potential tradeoff between L1 performance and L2 performance as a result of fusion, we determined the effects of fusing two loops in EXPL by several measures. Using reuse statistics available though GROUPPAD compiler analysis, we first determined the number of L2 references, i.e., the number of references in all loops which miss the L1 cache but hit the L2 cache, in the same manner as in Section 4. As in Section 4, we also determined the number of memory references, i.e., the number of references in all loops missing both the L1 and L2 cache.

L2 and memory references were computed assuming both GROUPPAD and L2MAXPAD transformations, so that all group reuse not exploited on the L1 cache was assumed to be preserved on the much larger L2 cache. L1 and L2 miss rates were then obtained before and after fusion. To account for a decrease in the reference count associated with fusion, miss rates for both versions were computed as the number of cache misses divided by the number of references in the *original* version. From this data we computed the *change* in L2 references, memory references, and cache miss rates as a result of fusion. These values were obtained for problem sizes ranging from 250 to 700.

[1] An improvement was found for DOT256, even though cache miss rates were not improved significantly. We believe this is due to the differences in the ability of the underlying memory system to handle multiple outstanding cache misses [1], since the two input vectors were padded 64 instead of 32 bytes due to the longer L2 cache lines. Such memory effects are prominent only for simple kernels such as DOT256, which have very few references. For larger loops the effects average out.

Results appear in Figure 12.

The upper graph reveals that the change in group reuse on the L1 cache may vary considerably depending on problem size. The increase in L2 references alternates between 1 and 2 before plateauing at 3. This high is maintained from problem size 334 to 398. After this point, fusion usually results in a small change in L2 references, 0 in most cases. Because of the larger capacity of the L2 cache, improved L2 locality as a result of fusion is full exploited, resulting in a constant decrease by 3 of the number of memory references. Thus, the upper graph suggests that the steady improvement in L2 performance as reflected by the memory reference count is offset somewhat by a loss of group reuse on the L1 cache, especially for problem sizes under 398.

The lower graph reveals a nearly linear relationship between the the computed references counts and the changes in cache miss rates as a result of fusion. The change in the L1 miss rate varies closely in proportion to the change in the number of L2 references. Like the change in memory references, the change in the L2 miss rate is a flat curve. However, the curve for L1 miss rates is not situated over the X-axis as is the curve for L2 references. Instead, this curve is somewhat lower, with the plateau from 334 to 398 barely breaking 0%. Thus, the miss rate improves on the L1 cache even with the net loss of group reuse on this cache. This is possibly due to an overall decrease in the number of L1 misses as the result of fusion. It is apparent though that had a fourth reference missed the L1 cache as the result of fusion, L1 performance would be adversely affected.

The results show that the compiler can predict relative cache miss rates fairly accurately by analyzing group reuse. As a result it should be able to accurately decide whether loop fusion is profitable, given the relative cost of L1 and L2 cache misses.

6.5 Tiling

Figure 13 compares UltraSparc performance in MFLOPS for different versions of matrix multiply. We examined matrices sizes from 100x100 to 400x400. None of the matrix sizes evaluated fit in L1 cache, and matrices larger than 256x256 do not fit in L2 cache. Several approaches for tile size selection are examined: picking L1-sized and L2-sized tiles (i.e., attempts to maximize L1 and L2 cache reuse, respectively), as well as picking intermediate sized tiles two and four times larger than the L1 cache (2xL1, 4xL1). We use the *eucPad* algorithm to choose tile dimensions which eliminate tile self-interference [23].

The graph shows the actual MFLOP rates for each version of the code. We see L1-sized tiles yield the best performance. They maximize L1 reuse but can also exploit L2 reuse, since performance is steady even for large matrices. In comparison, L2-sized tiles can improve performance for large matrices, but not small ones. The reason for this disadvantage is clear—L2-sized tiles are of no use when the data already fits in L2 cache.[2]

Intermediate tiles 2xL1 and 4XL1 achieve performance slightly higher than L2-sized tiles, showing most L1 benefits are lost as soon as tiles exceed what can fit in L1 cache. Results show that the benefits of exploiting L1 cache reuse outweigh the cost in capacity misses on the L2 cache. Choosing tile sizes based on L1 cache is thus effective in improving performance for both L1 and L2 cache, and yields best overall performance.

6.6 Discussion

Overall, our experiments show that while locality optimizations can be enhanced to improve miss rates multi-level caches, their actual impact on program performance is minimal. This outcome is because locality optimizations which target L1 caches also exploit most of the reuse at other levels of cache. As a result, existing optimizing compilers appear quite capable of achieving good performance for processors with multi-level caches.

7 Related Work

Data locality has been recognized as a significant performance issue for modern processor architectures. Wolf and Lam provide a concise definition and summary of important types of data locality [30]. Computation-reordering transformations such as loop permutation and tiling are the primary optimization techniques [8, 19, 24, 30], though loop fission (distribution) and loop fusion have also been found to be helpful [19]. Several cache estimation techniques have been proposed to help guide data locality optimizations [7, 8, 30]. Recently these techniques have been enhanced to take into account conflict misses due to limited cache associativity [10, 27].

Data layout optimizations such as padding and transpose have been shown to be useful in eliminating conflict misses and improving spatial locality [1, 13, 21, 22]. They have also been combined with loop transformations for improved effect [5, 12]. In previous work, we examined the applicability of inter-variable padding for eliminating severe conflict misses [21] and preserving group reuse [22].

[2] Raw MFLOP rates (around 38 MFLOPS) are about half of what is achieved by ATLAS [28], a tuned version of matrix multiply, on our UltraSparc (around 84 MFLOPS). However, if we unroll the loop by hand and apply scalar replacement, we achieve better performance (near 60 MFLOPS). The difference is thus mostly due to performance tuning to keep values in registers and better instruction scheduling. We believe our conclusions are still valid despite lower absolute performance.

In this paper we extend our padding algorithms to consider multi-level caches.

A number of researchers have examined techniques related to this paper. Manjikian and Abdelrahman propose a new loop fusion algorithm called *shift-and-peel* which expands the applicability of loop fusion [18]. They also propose *cache partitioning*, a version of MAXPAD which does not take severe conflict misses into account. Singhai and McKinley present a parameterized loop fusion algorithm which considers parallelism and register pressure in addition to reuse [25]. In comparison, our fusion algorithm explicitly calculates group reuse benefits for loop fusion in conjunction with inter-variable padding. We also consider multiple levels of cache.

Lam, Rothberg, Wolf show conflict misses can severely degrade the performance of tiling [17]. Coleman and McKinley select rectangular non-conflicting tile sizes [6] while others focus on using a portion of cache [32]. Chame and Moon propose a new cost model for estimating both interference and capacity misses to guide tiling [3]. Kodukula and Pingali develop *data shacking*, a data-centric approach to tiling which can be applied to a wide variety of loop nests, but doesn't account for tile conflicts [15, 16].

Song and Li extended tiling techniques to handle multiple loop nests [26]. They concentrate on only L2 cache since L1 cache is too small to provided reuse for their necessarily large tiles. Nonlinear array layouts can be used in conjunction with tiling to improve spatial locality [4].

In comparison to these previous researchers, we consider the effects of locality optimizations on multi-level caches. The only other researchers to target multiple levels of cache did so for tiling, while we expand the consideration of multi-level caches to other locality optimizations.

Mitchell *et al.* examined the interactions of multi-level tiling and effects for goals such as cache, TLB, and parallelism [20]. They found explicitly considering multiple levels of the memory hierarchy (cache and TLB) led to improved compromise tile sizes, since excessively wide tiles may result in TLB misses even if they fit in L1 cache. Because our cost model favors square tiles, our algorithms do not encounter this problem when picking L1-sized tiles.

Whaley and Dongarra developed ATLAS, an automatically tuned linear algebra library [28]. It performs tiling, relying on copying to avoid conflict misses. Tile size is selected based on the L1 cache size. However, the order tiles are executed may be affected by L2 cache size if panels of tiles do not fit. In such cases tiles are traversed with an additional level of tiling, yielding up to 20% improvements on the DEC Alpha 21164, which has a small 96K L2 cache. No improvement is found on other processors. The absolute performance of our tiled matrix multiply is lower than ATLAS because we are not as tuned for scalar replacement and instruction scheduling, but ATLAS supports our basic result that L1-sized tiles yield the best performance.

8 Conclusions

Compiler transformations can significantly improve data locality in scientific programs. In this paper, we show that most locality transformations can usually improve reuse for multiple levels of cache by simply targeting the smallest usable level of cache. Benefits for lower levels of cache are then obtained indirectly as a side effect. Some optimizations can benefit from explicitly considering multiple levels of cache, including inter-variable padding and loop fusion. Cache simulations and timings show while enhanced algorithms are able to reduce cache miss rates, they rarely improve execution times for current processors. While our results do not point out major opportunities to improve program performance, we believe the are still worthwhile because they indicate existing most compiler optimizations are sufficient to achieve good performance for multi-level caches.

References

[1] D. Bacon, J.-H. Chow, D.-C. Ju, K. Muthukumar, and V. Sarkar. A compiler framework for restructuring data declarations to enhance cache and TLB effectiveness. In *Proceedings of CASCON'94*, Toronto, Canada, October 1994.

[2] S. Carr and K. Kennedy. Compiler blockability of numerical algorithms. In *Proceedings of Supercomputing '92*, Minneapolis, MN, November 1992.

[3] J. Chame and S. Moon. A tile selection algorithm for data locality and cache interference. In *Proceedings of the 1999 ACM International Conference on Supercomputing*, Rhodes, Greece, June 1999.

[4] S. Chatterjee, V. Jain, A. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear array layouts for hierarchical memory systems. In *Proceedings of the 1999 ACM International Conference on Supercomputing*, Rhodes, Greece, June 1999.

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

[6] S. Coleman and K. 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.

[7] J. Ferrante, V. Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and*

Compilers for Parallel Computing, Fourth International Workshop, Santa Clara, CA, August 1991. Springer-Verlag.

[8] 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.

[9] G. Gao, R. Olsen, V. Sarkar, and R. Thekkath. Collective loop fusion for array contraction. In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992.

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

[12] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. Improving locality using loop and data transformations in an integrated framework. In *Proceedings of the 31th IEEE/ACM International Symposium on Microarchitecture*, Dallas, TX, November 1998.

[13] 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.

[14] K. Kennedy and K. S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[15] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multilevel blocking. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, Las Vegas, NV, June 1997.

[16] I. Kodukula and K. Pingali. An experimental evaluation of tiling and shacking for memory hierarchy management. In *Proceedings of the 1999 ACM International Conference on Supercomputing*, Rhodes, Greece, June 1999.

[17] 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.

[18] 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.

[19] 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.

[20] N. Mitchell, L. Carter, J. Ferrante, and K. Högstedt. Quantifying the multi-level nature of tiling interactions. In *Proceedings of the Tenth Workshop on Languages and Compilers for Parallel Computing*, Minneapolis, MN, August 1997.

[21] 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, June 1998.

[22] G. Rivera and C.-W. Tseng. Eliminating conflict misses for high performance architectures. In *Proceedings of the 1998 ACM International Conference on Supercomputing*, Melbourne, Australia, July 1998.

[23] G. Rivera and C.-W. Tseng. A comparison of compiler tiling algorithms. In *Proceedings of the 8th International Conference on Compiler Construction (CC'99)*, Amsterdam, The Netherlands, March 1999.

[24] V. Sarkar. Automatic selection of higher order transformations in the IBM XL Fortran compilers. *IBM Journal of Research and Development*, 41(3):233–264, May 1997.

[25] S. Singhai and K. S. McKinley. A parameterized loop fusion algorithm for improving parallelism and cache locality. *The Computer Journal*, 40(6):340–355, 1997.

[26] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999.

[27] 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.

[28] R. C. Whaley and J. Dongarra. Automatically tuned linear algebra software (ATLAS). In *Proceedings of SC'98*, Orlando, FL, November 1998.

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

[30] 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.

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

[32] M. E. Wolf, D. Maydan, and D.-K. Chen. Combining loop transformations considering caches and scheduling. In *Proceedings of the 29th IEEE/ACM International Symposium on Microarchitecture*, Paris, France, December 1996.

[33] M. J. Wolfe. More iteration space tiling. In *Proceedings of Supercomputing '89*, Reno, NV, November 1989.

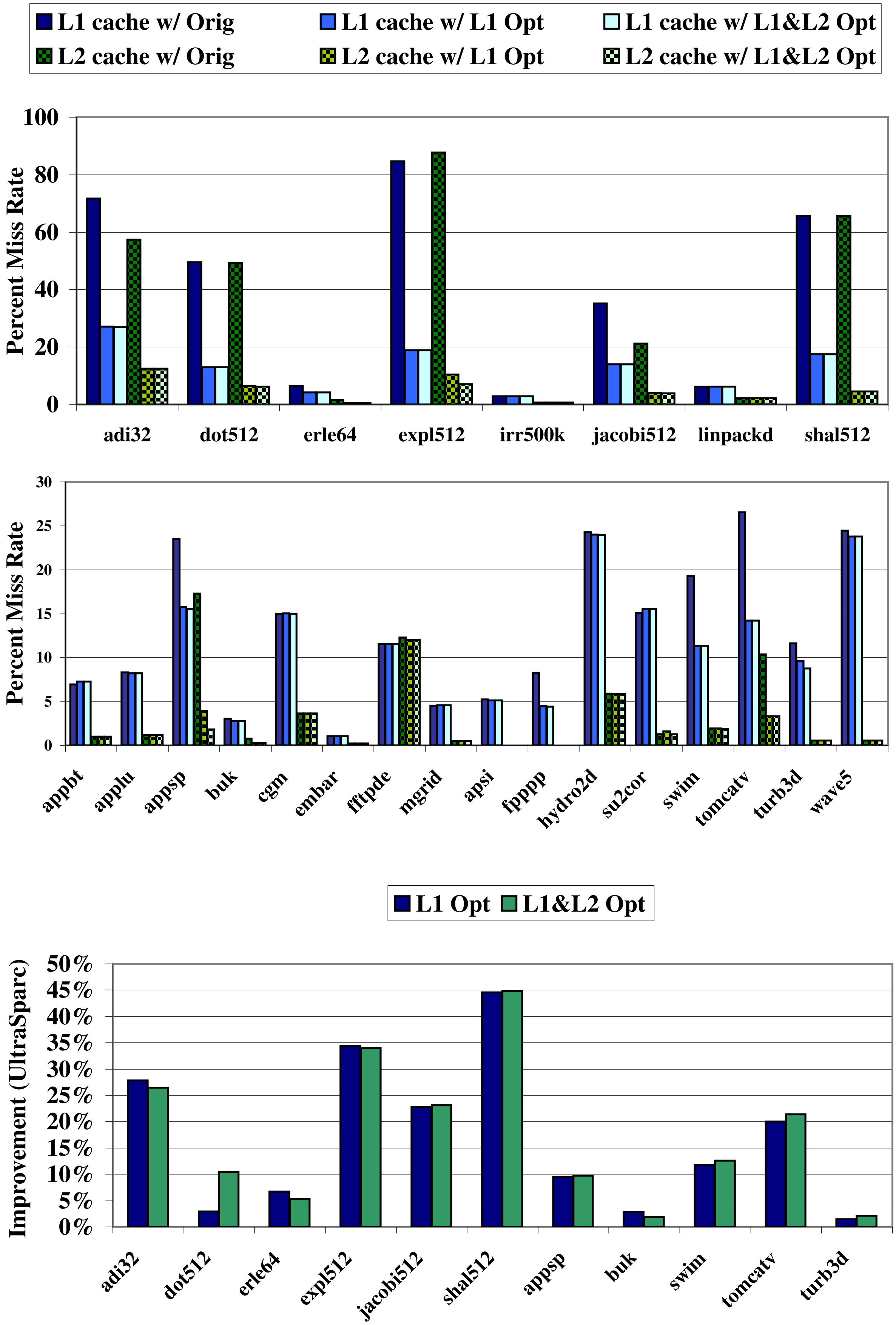


Figure 9 Miss rates and execution time improvements for PAD and MULTILVLPAD

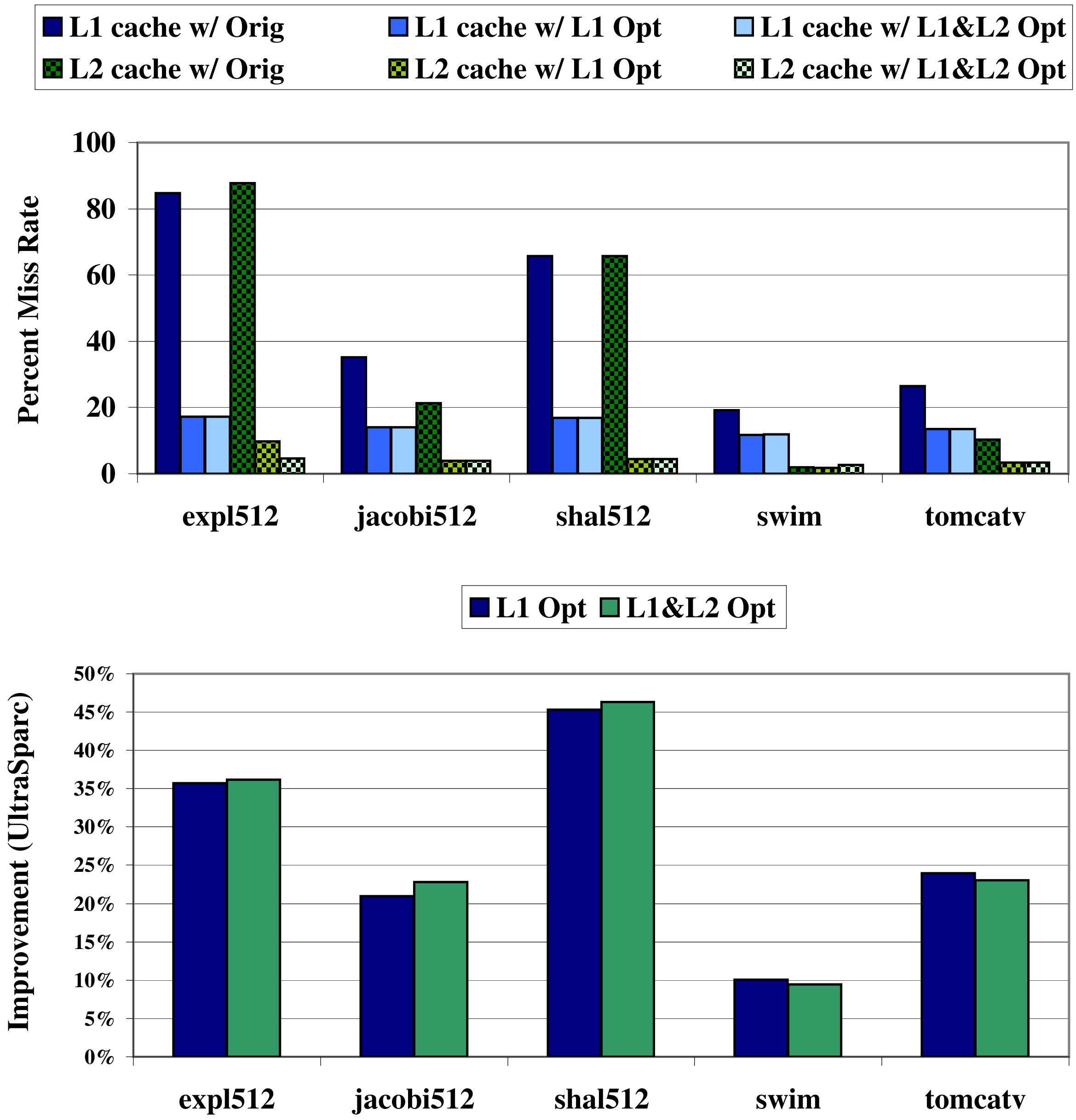


Figure 10 Miss rates and execution time improvements for GROUPPAD, with and without L2MAXPAD for optimizing the L2 cache

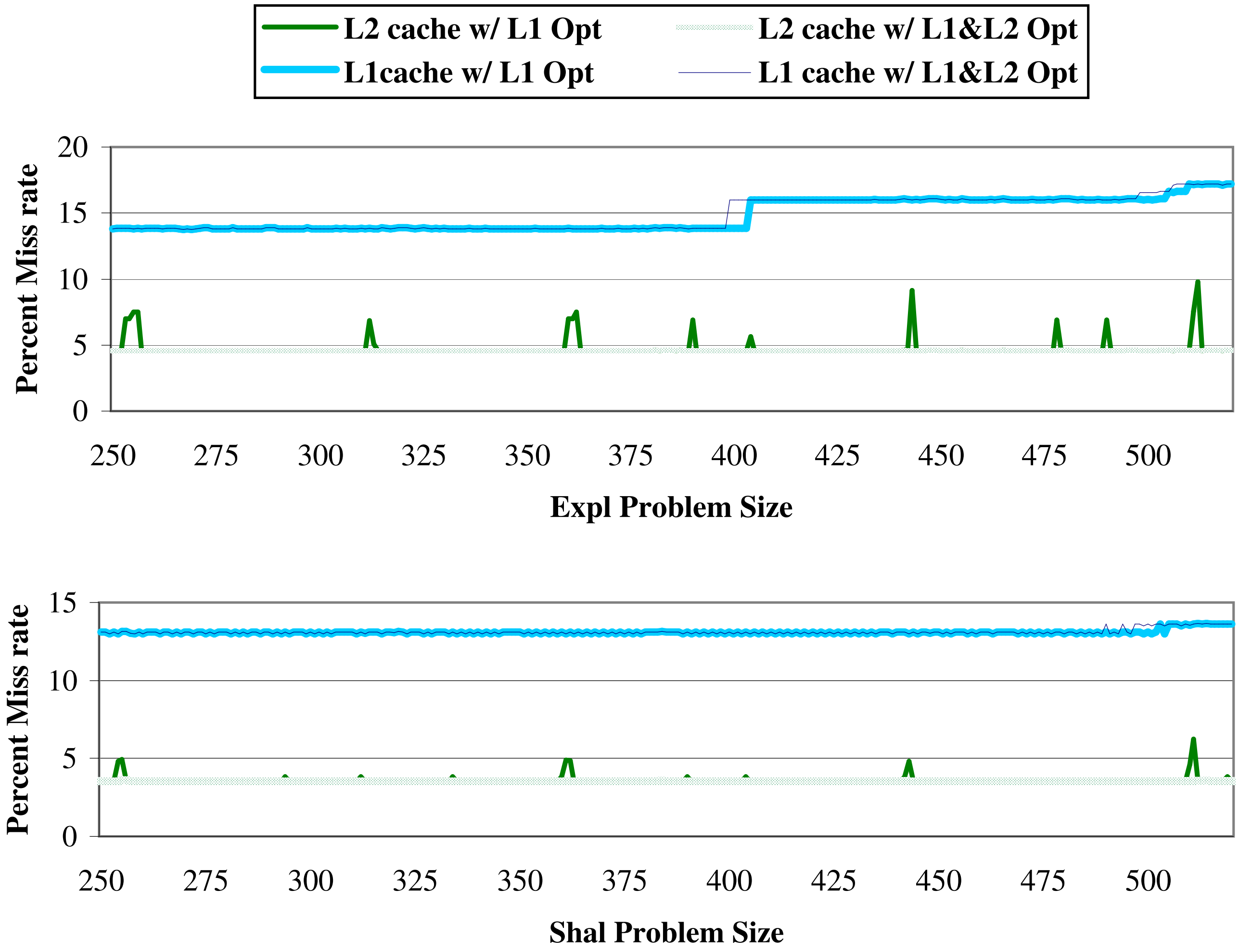
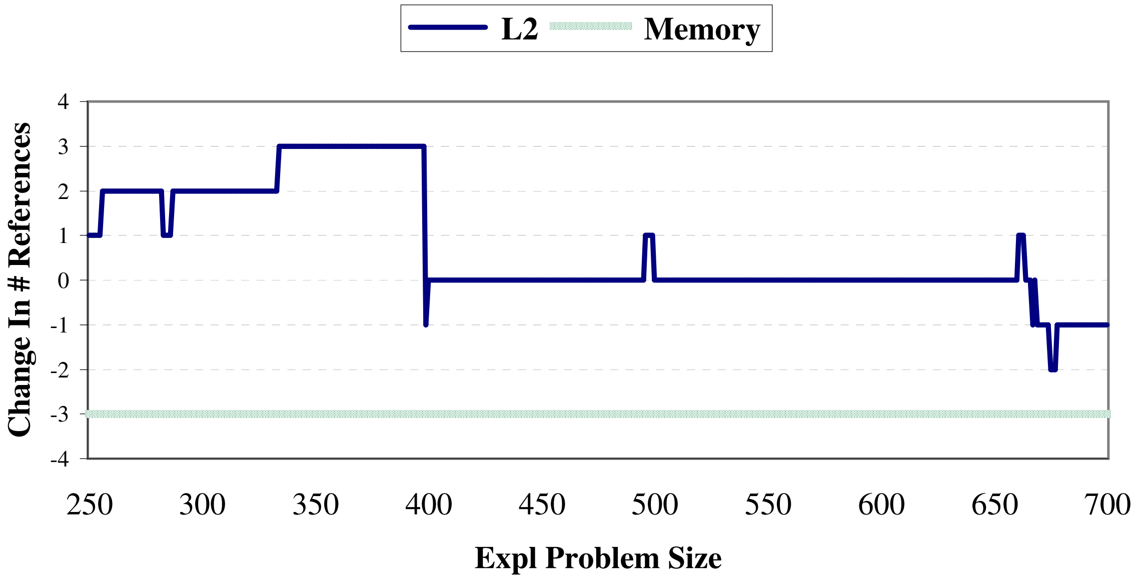


Figure 11 Cache miss rates over varying problem sizes for GROUPPAD with and without L2MAXPAD



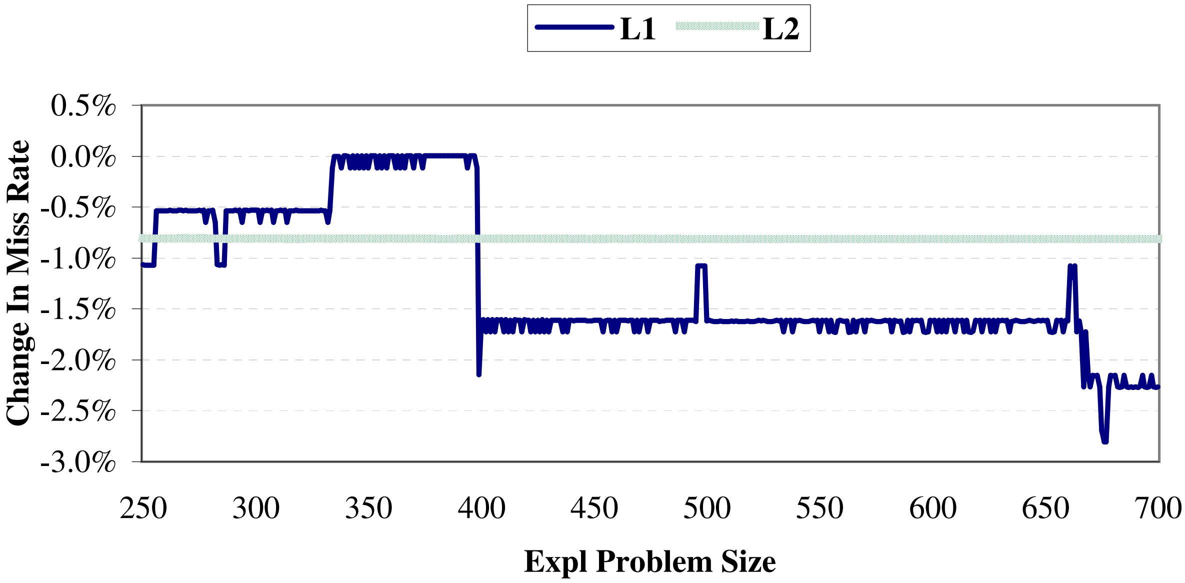


Figure 12 Change in L2 refs, memory refs, and miss rates as a result of fusion

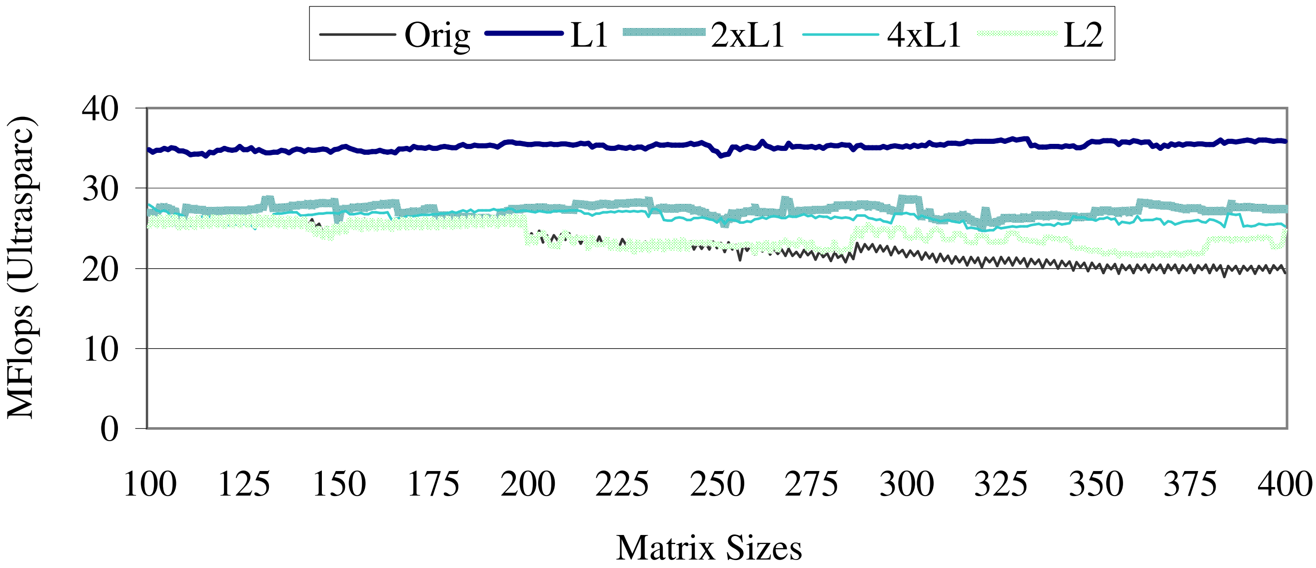


Figure 13 Performance for two tiling methods over varying problem size