Tiling Optimizations for 3D Scientific Computations

Gabriel Rivera, Chau-Wen Tseng

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

Abstract

Compiler transformations can significantly improve data locality for many scientific programs. In this paper, we show iterative solvers for partial differential equations (PDEs) in three dimensions require new compiler optimizations not needed for 2D codes, since reuse along the third dimension cannot fit in cache for larger problem sizes. Tiling is a program transformation compilers can apply to capture this reuse, but successful application of tiling requires selection of non-conflicting tiles and/or padding array dimensions to eliminate conflicts. We present new algorithms and cost models for selecting tiling shapes and array pads. We explain why tiling is rarely needed for 2D PDE solvers, but can be helpful for 3D stencil codes. Experimental results show tiling 3D codes can reduce miss rates and achieve performance improvements of 17–121% for key scientific kernels, including a 27% average improvement for the key computational loop nest in the SPEC/NAS benchmark MGRID.

```
A(N,N), B(N,N)
do J=2,N-1
 do I=2,N-1
  A(I,J) = C*(B(I-1,J)+B(I+1,J)+
              B(I,J-1)+B(I,J+1))
```

Figure 1 2D Jacobi iteration code

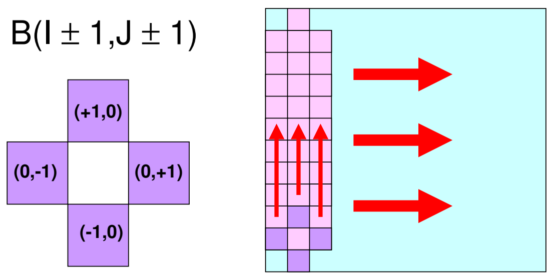

Figure 2 Access pattern for 2D Jacobi

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. 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 through loop transformations (e.g., loop permutation, fusion, fission) [21, 33], or change the layout of data through data transformations (e.g., padding, transpose) [1, 16, 24]. Experiments show compilers can improve the performance of many benchmark programs, some times dramatically.

An important class of scientific programs attempt to compute solutions to partial differential equations (PDEs) using finite differencing techniques. Figure 1 presents the code for Jacobi iteration, a simple PDE solver. These solvers are also called *stencil* codes because they compute values using neighboring array elements in a fixed stencil pattern. This stencil pattern of data accesses is then repeated for each element of the array.

For instance, the Jacobi iteration kernel consists of a simple 4-point stencil in two dimensions, shown in the first part of Figure 2. On each loop iteration, four elements of the array are accessed in the 4-point diamond stencil pattern shown on the left. As the computation progresses, the stencil pattern is repeatedly applied to array elements in the column, sweeping through the array, as shown in the second part of Figure 2.

Historically PDE solvers have targeted 2D domains. As computers became more powerful, scientists have begun writing programs to solve PDEs in three dimensions. For instance, Figure 3 presents a 3D Jacobi iteration solver. The code uses a 6-point stencil in three dimensions, shown in Figure 4.

0-7803-9802-5/2000/$10.00 (c) 2000 IEEE.
In Proceedings of SC'00, November 2000, Dallas, TX.

```
A(N,N,N), B(N,N,N)
do K=2,N-1
 do J=2,N-1
  do I=2,N-1
   A(I,J,K) = C*(B(I-1,J,K)+B(I+1,J,K)+
                 B(I,J-1,K)+B(I,J+1,K)+
                 B(I,J,K-1)+B(I,J,K+1))
```

Figure 3 3D Jacobi iteration code

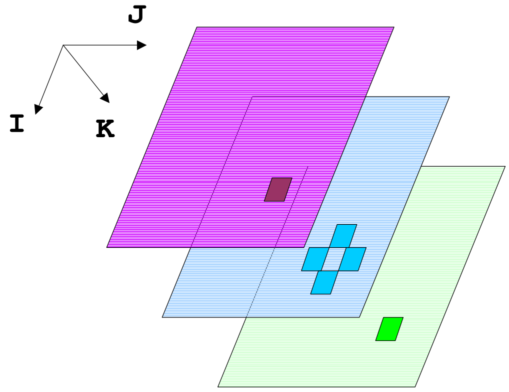

Figure 4 Access pattern for 3D Jacobi

As 3D stencil codes become widespread, numerical analysts discover that they have particularly poor memory behavior with respect to microprocessor caches [32]. Ideally, applications can minimize cache misses by bringing data into cache just once for all of its multiple accesses. 3D PDE solvers suffer poor cache performance because accesses to the same data are usually too far apart, requiring array elements to be brought into cache multiple times per array sweep.

This cache problem arises more often in 3D codes than in 2D codes. To see why, compare the 2D Jacobi code in Figure 1 with the 3D code in Figure 3. In the 2D Jacobi kernel, a 4-point stencil sweeps across array B in unit stride, accessing at once a total of three columns of B. Note that the distance in memory between the leading reference `B(I,J+1)` and the trailing reference `B(I,J-1)` is 2N, where N is the column size, assuming column-major order. Therefore, the cache only needs to be able to hold two columns of B to support all group reuse between the B references (assuming a write around cache, so A does not interfere). Even a relatively small 16K primary (L1) cache (which holds 2048 double precision words) can hold all the data needed up to a 1024×M array of doubles (where M can be any value). For 2D arrays with column size less than 1024, values of B only need to be brought into cache once as the kernel sweeps across B.

In comparison, the 3D Jacobi kernel has a 6-point stencil which accesses six columns of B in three adjacent planes at the same time, as shown in Figure 4. With a distance of $2N^2$ between the leading `A(I,J,K+1)` and trailing `A(I,J,K-1)` array references, two entire N×N planes now need to remain in cache, so only 3D arrays of size 32×32×M can fully exploit reuse for a 16K L1 cache. Even for a larger 2M secondary (L2) cache, group reuse is lost for 3D arrays larger than 362×362×M. Data for array B will need to be brought into cache two or more times each time the kernel is executed, reducing performance for larger problem sizes.

Tiling is a well known transformation which improves locality by moving reuses to the same data closer in time. As we show in later sections, existing methods such as the algorithm of Wolf and Lam [33] can have only a minor impact on 3D stencils. Approaches devised specifically for stencils such as the method of Song and Li [29] are neither directly applicable in the 3D case nor extensible to more specialized multigrid applications.

In this paper we examine the problems inhibiting locality for 3D stencil codes. We show that tiling can significantly improve reuse, especially when combined with array padding. The contributions for this paper are:

- Showing why tiling is not needed for 2D stencil codes.
- New tiling and padding transformations for 3D stencil codes.
- Experimental evaluation of tiling transformations on 3D stencil kernels and the SPEC/NAS multigrid benchmark.

The remainder of this paper is arranged as follows: Sections 2 and 3 present tiling and padding transformations for 3D stencils. Section 4 evaluates our transformations through cache simulations and actual performance measurements and discusses our results. Section 5 describes related work and the final section states our conclusions.

2 Tiling Transformations

In this section we examine tiling transformations for 3D stencils. Tiling (blocking) is a transformation which combines strip-mining with loop permutation to form small tiles of loop iterations which are executed together to exploit data locality [2, 15, 36].

2.1 Tiling for stencil codes

Tiling has been shown to be very effective for linear algebra codes. Because they perform $O(n^3)$ computations over $O(n^2)$ data, tiling can exploit $O(n)$ temporal reuse to greatly reduce cache misses. Tiling can also yield large

```
// Simplified Stencil Code
do T = 1,time
 do J=2,N-1
  do I=2,N-1
    A(I,J) = B(I-1,J)+...

// Realistic Stencil Code
do T = 1,time
 do J=2,N-1
  do I=2,N-1
    A(I,J) = B(I-1,J)+...
 do J=2,N-1
  do I=2,N-1
    B(I,J) = A(I,J)
 ...

// Multigrid Code
do T = 1,time
 do S = 0,lg(N)-1
  N = 2^S
  do J=2,N-1
   do I=2,N-1
     B(I,J) = A(I-1,J)+...
```

Figure 5 Stencil codes

benefits for simple stencil kernels with a time-step loop enclosing a single stencil computation, an example of which appears first in Figure 5. These stencil codes can be found in benchmarks like the the Livermore loops, and yield large improvements since they also perform $O(n^3)$ computation over $O(n^2)$ data. To exploit temporal reuse, compilers can simply skew the time-step loop with respect to the inner stencil loops [33, 6, 3, 23].

Unfortunately, we believe these simple stencil kernels *are in fact over-simplified*. A more realistic stencil code would actually have multiple loop nests within the time-step loop, in order to actually compute useful values. An example of a more realistic stencil code is shown in the middle of Figure 5. This pattern is found in programs such as TOMCATV, SWIM, APPBT, and APPSP from the SPEC and NAS benchmark suites.

Simple skewing of tiles is not possible with multiple loop nests. To exploit temporal reuse from the time-step loop, the compiler must perform much more complex analyses and transformations to match and fuse tiles from the multiple loop nests. Such techniques have been developed for 2D arrays [29, 37], but not for 3D. Instead, most optimizations have focused on exploiting temporal and spatial reuse within individual loop nests [21, 33]. Tiling is usually not needed, since most locality can be obtained through loop permutation, though in some cases array padding may be necessary to preserve group reuse [25].

Finally, we consider multigrid codes, a third type of stencil code where even these locality transformations are not possible. *Multigrid* codes are also iterative PDE solvers, but speed up convergence on a solution by using a succession of grids mapped on top of each other. For easier mapping, grids are usually chosen to be powers of two. An example multigrid code is shown in the final code in Figure 5. In addition to the original stencil and time-step loops, there is an additional loop which iterates over multiple grid sizes on each time step. This additional loop is what prevents even the more powerful transformations [29, 37] from exploiting reuse on the time-step loop.

The tiling techniques we present in this paper are designed to exploit group-temporal reuse in both realistic and multigrid 3D stencil codes. They do not exploit large amounts of temporal reuse on the time-step loop, but can still yield reasonable performance improvements. In the future we hope to combine our techniques with theirs to generate non-conflicting time-skewed stencil computations.

2.2 Selecting Tile Sizes

Our basic transformation tiles as few loops as needed to preserve group reuse in 3D loop nests. Section 1 described how the size of the array plane affects whether group reuse is preserved across the outermost loop (referred to as the *K loop* in accordance with Figure 4.) The goal of our tiling method is thus to enable this reuse by effectively reducing the size of the plane during the execution of the K loop. This is accomplished by *tiling* the inner two (J and I) loops. First, J and I are strip-mined to form tile-controlling loops JJ and II. Next, JJ and II are permuted to the outermost level. A tiled version of 3D Jacobi from Figure 3 appears in Figure 6.

Achieving reuse after applying this basic tiling transformation depends on the choice of tile dimensions TI and TJ. To motivate this decision, Figure 7 illustrates the data access patterns of the tiled version of 3D Jacobi. Since we tile only the J and I loops, the K loop iterates across all array planes but executes only iteration points inside a TI$\times$TJ$\times$(N-2) block (delineated with dashed lines in the figure). The shaded region represents the points in *iteration* space accessed on a single K loop iteration while the surrounding three unshaded regions represent the *array* points of B accessed on a single K loop iteration. We refer to the prior region as an *iteration* tile and the latter region as an *array* tile. Because of the stencil pattern found in Jacobi is simply ± 1 in each dimension, the array tile consists of two TI$\times$TJ regions located in array planes K-1 and K+1 as well as a third (TI+2)$\times$(TJ+2) region located in array plane K. In order to preserve all reuse within the TI$\times$TJ$\times$(N-2) block, it is therefore sufficient that the cache hold a (TI+2)$\times$(TJ+2)$\times$3 subarray, as long as tiles

```
do JJ=2,N-1,TJ
 do II=2,N-1,TI
  do K=2,N-1
   do J=JJ,min(JJ+TJ-1,N-1)
    do I=II,min(II+TI-1,N-1)
     A(I,J,K) = C*(B(I-1,J,K)+B(I+1,J,K) +
                   B(I,J-1,K)+B(I,J+1,K) +
                   B(I,J,K-1)+B(I,J,K+1))
```

Figure 6 Tiled 3D Jacobi iteration

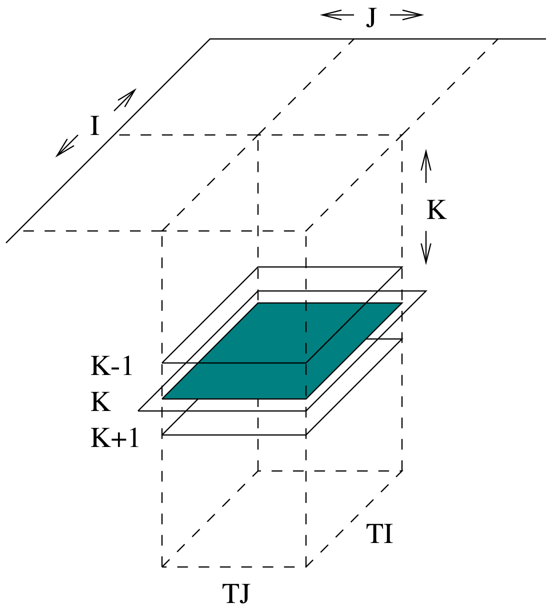

Figure 7 Access pattern of tiled 3D Jacobi

are executed in order along the K dimension. Thus, unlike many linear algebra codes, for which we also tile only two loops, 3D stencil codes require that the cache hold array tiles with three (not 2) dimensions to preserve reuse.

Existing tiling algorithms such as that of Wolf and Lam [33] will tile all three loops because reuse is carried across all the loops in 3D stencils. However, we have shown that tiling only two loops is sufficient for preserving reuse across all three loops. This follows from the fact that in 3D stencils we merely need to preserve group reuse with a distance of three. In contrast, preserving the self-temporal reuse common in linear algebra codes requires tiles with the same dimensionality as the arrays. Unlike tiling only two loops in 3D stencils, tiling three loops has the effect of increasing the number of tiles executed, leading to an additional loss of reuse along expanded tile boundaries.

Another approach introduced by Song and Li [29] preserves reuse across the time step loops which enclose the stencil loops in iterative PDE solvers. However, their technique does not extend to the 3D stencils which form the computational core of multigrid solvers, since these applications use a succession of grid sizes to speed up convergence. In contrast, we can apply the basic tiling transformation introduced here to multigrid solvers since only the 3D stencil loop is affected.

2.3 Cost function

As stated in the previous section, the dimensions of the iteration tile (*tile size*) must be selected in order for the array tile (e.g., 3 TI$\times$TJ planes) to fit in cache. In this section we see that even among tile sizes which meet this constraint, some lead to better reuse than others. To determine the tile size (TI, TJ) which best preserves reuse, we estimate the number of cache misses that arises from a tile size, favoring tile sizes which result in fewer cache misses. We compute the number of cache misses simply as the number of distinct cache lines accessed in each TI$\times$TJ$\times$(N-2) block of iterations. To accomplish this, we first make the reasonable assumption that data accessed in each block is not initially in cache. Another critical assumption is that all given tile sizes lead to a non self-interfering array tile. Later in Section 3, we discuss how to obtain such non self-interfering tile sizes.

We use the example in Figure 6 to motivate a formula for the number of cache misses. During each TI$\times$TJ$\times$(N-2) block of iterations, we access approximately (TI+2)(TJ+2)N elements of array B. While this formula depends on the particular access pattern, loop nests in 3D PDE solvers will generally access about (TI+m)(TJ+n)N elements where m and n merely depend on the particular stencil pattern (set by the magnitude of the largest differences between subscripts in each dimension).

Since the iteration space is partitioned into approximately $N^2/$(TI$\times$TJ) blocks of size TI$\times$TJ$\times$(N-2), we multiply to obtain the total number of elements brought into cache across the whole loop: N^3(TI+2)(TJ+2)/(TI$\times$TJ). We can then divide by the cache line size L to estimate the number of cache lines fetched. However, since N^3/L is invariant under different tiled sizes, we divide out this constant, resulting in the function $Cost$(TI,TJ) = (TI+2)(TJ+2)/(TI$\times$TJ). Note that given multiple tile sizes with equal values of TI$\times$TJ, this function is minimal when TI and TJ have the smallest difference. The cost function thus favors square tiles.

Using such techniques, compilers can derive such a cost function directly from the loop nest to model the loss of reuse (i.e., the cost) for each tile size. In Section 3 we present tiling transformations which select the tile size based on this cost function.

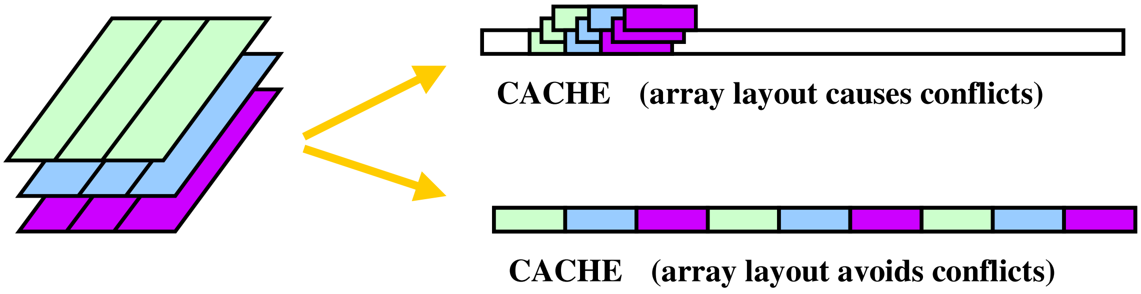

Figure 8 Example of intra-tile conflict misses under two array layouts

3 Avoiding Conflicts

In this section we describe *conflict misses*, show how they can limit the effectiveness of tiling, and present several approaches for eliminating conflicts. Conflict misses arise because caches have limited *set associativity*, where memory addresses map only to one of n locations in a n-way associative cache. They 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 overall sufficient capacity in the cache. Conflict misses have been shown to severely degrade the performance of tiled codes [20].

In Section 2 we found that to improve reuse, the array tile must fit in cache and the tile size should be favorable according to the cost function. In this section we also show that the array tile should avoid self-interference. The number of conflicts inside the array tile is affected by both the tile size and the array dimensions. Intuitively, the effect of the tile size on conflicts should be clear since expanding the tile size can introduce conflicts. To illustrate the impact of the array dimensions on conflicts, Figure 8 shows how the array tile resulting from our tiling transformation might map onto a direct-mapped cache under two different array layouts. Note that the columns of each plane of the array tile are represented as a single block since they are arranged contiguously in memory and cannot conflict with themselves. Conflicts occur when different columns overlap with each other in cache, as shown in the upper data mapping in Figure 8. Under a change in array dimensions, the data can distribute more evenly over the cache, as seen in the lower data mapping.

Four methods have been proposed for eliminating conflicts between elements of a tile: copying, using a small fraction of the cache, tile size selection, and padding. We consider each approach, developing several strategies and heuristics which we then evaluate in Section 4.

3.1 Copy optimization

Copying tiles into contiguous buffers is one method for avoiding conflicts [20, 26, 31]. It works well for linear algebra codes because each tile can be reused a large number of times, amortizing the overhead of performing the copy. In matrix multiplication for instance, copying costs are asymptotically negligible since $O(N^2)$ elements are copied in a tiled loop nest performing $O(N^3)$ data accesses. In stencil codes however, reuse is much less. Copying tiles is not possible without copy operations comprising a large, *constant* fraction of the data accesses. Copying is therefore not profitable for stencil codes.

3.2 Using a subset of the cache

A second method called *effective cache size* estimates conflicts which may arise in cache, then computes the size of a subset of cache, which is a small fraction of the actual cache size [28, 34]. The compiler then simply chooses smaller tiles that target the effective cache size. Experimental evaluations seem to indicate the effective cache size is close to 10% for tiled codes [26, 34]. This method thus has the disadvantage of not fully utilizing the cache. Additionally, for pathological array dimension sizes which divide or nearly divide the cache size, many conflicts may still result despite choosing smaller tiles [26].

3.3 Tile size selection (Euc3D)

A third method for avoiding conflicts is to carefully select tile dimensions tailored to the particular array dimensions so that no conflicts occur. For 2D arrays, simple recurrences based on the Euclidean remainder algorithm may be used to quickly compute a sequence of non-conflicting tile dimensions [6, 26]. A cost function is used to select the tiles preserving the most reuse. A search space algorithm using a very precise cache model can obtain similar results [12, 14].

To efficiently compute non-conflicting tile dimensions for 3D arrays we introduce Euc3D, an extension to the Euc algorithm given in [26]. The pseudocode in Figure 9 presents an overview of Euc3D. Like Euc, the algorithm initially computes several non-conflicting array tiles and then selects from among them the tile minimizing the cost

TK	1	1	1	1	2	2	2	2	3	3	3	4	4	4	
TJ	1	10	41	256	1	4	5	15	5	11	15	4	15	56	...
TI	2048	200	48	8	960	200	160	40	72	40	24	72	16	8	

Table 1 Euc3D example non-conflicting tile sizes (200×200×M array, 16K cache)

Euc3D(C_s, DI, DJ) {
  $(TI_{mc}, TJ_{mc}) \leftarrow (1, 1)$
  for $TK = 1 \ldots ATD$
    [Run Euclidean algorithm to compute
     non-conflicting array tiles with depth TK
     for DI×DJ×M array]
  foreach resulting array tile (TI, TJ, TK)
    $(TI', TJ') \leftarrow (TI - 2, TJ - 2)$
    if $Cost(TI', TJ') < Cost(TI_{mc}, TJ_{mc})$ then
      $(TI_{mc}, TJ_{mc}) \leftarrow (TI', TJ')$
  return (TI_{mc}, TJ_{mc}) }

Figure 9 Euc3D pseudocode. Computes tile size (TI_{mc}, TJ_{mc}) from cache size C_s and lower array dimensions (DI, DJ).

function. To compute the array tiles, the *for* loop runs the Euclidean algorithm ATD times, where ATD is the minimum *array tile depth*, the number of planes (e.g., 3 for Jacobi) in the array tile we intend to keep in cache. Each new *for* loop iteration combines new computation with results from the previous iteration to find array tiles with depth TK. (We omit some details of the Euc3D algorithm in Figure 9 to simplify the presentation.)

Next, the *foreach* loop iterates through the array tiles computed in the *for* loop with TK $\geq$ ATD to compute the iteration tile which minimizes the cost function. The iteration tile dimensions TI' and TJ' are computed based on the array tile dimensions TI and TJ and the access patterns of the tiled loop nest. For instance, for the 3D Jacobi nest in Figure 6, array tile dimensions are *trimmed* by 2 or else the iteration tile will exceed the boundaries of the array tile through accesses to `B`. (Note that writes to `A` will not affect the cache given a write-around cache.) Trimming of both TI and TJ by 2 is illustrated in Figure 9. After trimming, any tile with nonpositive TI' or TJ' is discarded, accomplished by having the $Cost$ function return infinity in these cases. Finally, (TI',TJ') will then replace the current minimum cost tile (TI_{mc},TJ_{mc}) if its $Cost$ is smaller.

Table 1 illustrates several non-conflicting array tiles given a 200×200×M array (where M can be any value) and a (16K) cache which holds 2048 array elements. Several additional array tiles with TK > 4 are omitted. The *foreach* loop in Figure 9 considers each array tile with with TK $\geq$ ATD, selecting the one minimizing the cost function. For instance, after trimming the array tile sizes by 2 in each dimension, the cost function (TI+2)(TJ+2)/(TI×TJ) is used to select the final minimum cost tile (TI_{mc},TJ_{mc}) = (22,13) which originates from the array tile with TK=3, TJ=15, TI=24.

The running time for Euc3D is very low. Since the Euclidean algorithm requires only a logarithmic number of steps to complete, each step of *for* loop executes in time $O(log(C_s))$, where C_s is the cache size. Since we generally supply a small, constant ATD (e.g., 3) when tiling 3D stencil codes, the complexity for the entire *for* loop is $O(log(C_s))$. The complexity for the overall algorithm is therefore the same since the number of tile sizes considered in the *foreach* is limited by the number of array tiles computed in the *for* loop.

Efficiency in choosing tile sizes is not very important for arrays of fixed size, since tile sizes can be chosen at compile time. However, inexpensive algorithms can have an impact on codes where array sizes are not known at compile time, such as multigrids codes where different size grids are computed at each invocation. The efficiency of the Euc3D algorithm is thus attractive compared earlier algorithms such as the tile size selection method proposed by Lam, Rothberg, and Wolf [20] which has complexity O($\sqrt{C}$) and does not handle 3D arrays.

3.4 Padding

Even when choosing non-conflicting tiles, performance of tiling may suffer for certain array dimensions. In such cases, non-conflicting tiles result where one or more dimensions are pathologically small, reducing the benefits of tiling. For instance, given a 341×341×M array, the best tile size available is (110, 4). One possible solution is to use padding to enable better tile sizes [1, 24]. Padding increases the size of leading array dimensions, increasing the range of non-conflicting tile shapes. It has proven to be very useful for improving tiling for 2D linear algebra codes [26].

3.4.1 Padding for fixed tile size (GcdPad)

We describe two heuristics for applying padding. The first, GcdPad, applies padding to eliminate conflicts after first selecting a tile size. An advantage to this method is its very low compiler overhead. A disadvantage is the amount of padding overhead required. The method first computes non-conflicting array tile dimensions, next

GcdPad (C_s, DI, DJ) {
  $TK \leftarrow 4$
  $TI \leftarrow 2^{\lceil \log_2 \sqrt{C_s/TK} \rceil}$
  $TJ \leftarrow C_s/(TK\, TI)$
    $(TI', TJ') \leftarrow (TI-2, TJ-2)$
  $DI_p \leftarrow 2TI\lfloor (DI+3TI-1)/(2TI) \rfloor - TI$
  $DJ_p \leftarrow 2TJ\lfloor (DJ+3TJ-1)/(2TJ) \rfloor - TJ$
  return (TI', TJ', DI_p, DJ_p) }

Figure 10 GcdPad pseudocode. Computes tile size (TI', TJ') and padded lower array dimensions (DI_p, DJ_p) from cache size C_s and original lower array dimensions (DI, DJ).

Pad (C_s, DI, DJ) {
  $(TI_g, TJ_g, DI_g, DJ_g) \leftarrow$ GcdPad (C_s, DI, DJ)
  $Cost^* \leftarrow Cost(TI_g, TJ_g)$
  for $DI_p = DI \ldots DI_g$
   for $DJ_p = DJ \ldots DJ_g$
    $(TI', TJ') \leftarrow$Euc3D(C_s, DI_p, DJ_p)
    if $Cost(TI', TJ') \leq Cost^*$ then
     return (TI', TJ', DI_p, DJ_p) }

Figure 11 Pad pseudocode. Parameters and return values are the same as for GcdPad.

computes the actual tile dimensions for the nest, and finally determines the padded array dimensions for eliminating self-interference in the array tile. The method is based on the observation that when the following conditions are satisfied, non-conflicting array tiles result:

- The product of all of the tile dimensions is equal to or less than the cache size (C_s)
- For each array dimension with the exception of the highest-order dimensions the greatest common divisor (gcd) of the array dimension size and C_s is equal to the corresponding tile dimension

For example, for 3D tiles, we require that (TI$\times$TJ$\times$TK) $\leq C_s$ and that assuming a DI$\times$DJ$\times$DK array, $gcd(\text{DI}, C_s) =$TI and $gcd(\text{DJ}, C_s) =$TJ.

The tile size and array dimensions are computed as shown in the GcdPad pseudocode given in Figure 10. The algorithm first computes TI, TJ, and TK each as factors of the cache size (powers of two) whose product equals the cache size. TK is normally chosen as 4 since only 3-4 tile planes must exist in cache depending on the target tiled nest. Next TI and TJ are chosen such that TI$\times$TJ$\times$TK=C_s. This is done by setting TK to 4, setting TI to the smallest power of two greater than or equal to $\sqrt{C_s/TK}$, and then simply setting TJ to $C_s/$(TK$\times$TI). For example, if $C_s = 2048$ (array elements), GcdPad chooses (TI,TJ,TK)=(32,16,4). This strategy results in tiles which are large enough in both the I and J dimensions to adequately minimize the cost function.

Next, the iteration tile dimensions TI' and TJ' are computed from the array tile dimensions by trimming TI and TJ as discussed in Section 3.3. (Figure 10 again shows the case where both dimensions are trimmed by 2). Finally, the padded array dimensions DI$_p$ and DJ$_p$ are computed from the original dimensions DI and DJ such that $gcd(\text{DI}_p, C_s) =$TI and $gcd(\text{DJ}_p, C_s) =$TJ. Returning to the case where $C_s = 2048$, this requires padding DI at most 2 TI $-1 = 2 \times 32 - 1 = 63$ and DJ by at most 2 TJ $-1 = 2 \times 16 - 1 = 31$. For instance, when 224 <DI< 288, DI$_p$ is set to 288, an increment of DI by at most 63. Similarly, in the next 64-interval, DI$_p$ is set to 352.

3.4.2 Padding with tile size selection (Pad)

Our second padding heuristic, Pad, extends Euc3D for multiple array dimensions. Euc3D is run over a small set of padded array sizes, resulting in a tile size for each array size. Once a tile is found which meets a cost threshold $Cost^*$ selected in advance, the array dimension producing that tile size and the tile size itself are selected by Pad.

Pseudocode for Pad is given in Figure 11. This code makes use of the GcdPad function given in Figure 10 as well as the Euc3D function given in Figure 9. The values returned by GcdPad are used to bound the search for array pads and to choose $Cost^*$ as follows: Pad calls GcdPad to obtain the GcdPad tile size (TI$_g$,TJ$_g$) and the GcdPad lower array dimensions DI$_g$ and DJ$_g$. Pad then determines the value $Cost^*$ substituting TI$_g$ and TJ$_g$ into the cost function. This ensures that the tile size selected by Pad has cost at least as small as that of GcdPad. Finally, Pad runs Euc3D for array sizes with dimensions smaller than or equal to the GcdPad dimensions. (Note that tiles returned by Euc3D are already trimmed). The first tile found with cost $\leq Cost^*$ is selected along with the lower array dimensions DI$_p$ and DJ$_p$ which enabled the tile. Note that a tile with cost $\leq Cost^*$ must be found since the search space includes the tile (TI$_g$,TJ$_g$) returned by GcdPad.

Compared to GcdPad, the padding overhead required of Pad is always equal or smaller since the array dimensions selected by GcdPad are used to bound the padding search. However, a disadvantage to Pad is the added complexity and compiler overhead (though still very small in practice).

3.5 Cross-interference

In more complicated kernels, data accesses to multiple arrays lead to *cross-interference* misses. Several strategies for dealing with cross-interference are possible. If the number of cross-interfering references is relatively small, one option is simply to tolerate cross-interference, proceeding as if these references were not present. As we see in Section 4, this works well with kernels such as the RESID subroutine shown in Figure 13, where a single V reference can interfere with reuse between U references. Here, the benefits of tiling are still significant in spite of interference from V since there is so much group reuse of U.

A second potential strategy is to proceed with Euc3D, GcdPad, or Pad, obtaining a non-conflicting array tile, and then partition the array tile between arrays. This can be accomplished by reducing one tile dimension and then applying *inter-variable padding* so that each array accesses data mapping to its own portion of the array tile [25].

4 Experimental Evaluation

4.1 Benchmarks

To examine the benefits of tiling 3D PDE solvers and compare the effectiveness of the tiling and padding transformations introduced earlier, we evaluated our transformations on three kernel benchmarks.

Jacobi (JACOBI) The first benchmark is the 3D Jacobi iteration kernel we used as an example in Figure 3. It computes a simple six-point finite-difference stencil in three dimensions to solve 3D PDEs.

Red-black SOR (REDBLACK) REDBLACK is another iterative PDE solver similar to JACOBI, but uses Red-black Successive-Over-Relaxation (SOR) instead.

REDBLACK first accesses all "red" array points (where sum of coordinates is even) to compute values for "black" array points (where sum of coordinates is odd), then accesses all black points to compute values for red points. REDBLACK is a common component in multigrid codes; a 3D version is shown in the top example of Figure 12. One advantage of REDBLACK compared to JACOBI is that a second array is not needed to store new computed values. Instead, results can be directly stored in the array, since it is of the opposite color.

A disadvantage of REDBLACK is that in a standard implementation, array data will be brought into cache multiple times if the array size exceeds the cache size (once for red points, once for black points, more if multiple loops are used for each color), as shown in Figure 12. In addition, under standard memory storage only half of each cache line is utilized for each color.

Researchers have shown how to avoid this problem (in the 2D case) by ordering loop iterations so that black points in each column are updated immediately after the red points in the next column, and vice versa [32]. In the improved 2D version, all reuse is preserved provided three columns fit in cache.

In the 3D case, we can similarly update black points in plane K-1 after updating red points in plane K by fusing the red and black loop nests, as shown in the second example in Figure 12. The fused version of REDBLACK is then similar to 3D Jacobi: *three* entire array planes must fit in cache or else A will be brought into cache multiple times. We can then apply tiling to the fused loop nest to preserve group-reuse, resulting in the bottom example of tiled REDBLACK in Figure 12.

```
// Naive Version
do odd=0,1
 do K=2,N-1
  do J=2,N-1
   do I=2+mod(K+J+odd,2),N-1,2
    A(I,J,K) = C1*A(I, J, K) + C2 *
        ( A(I-1, J, K) + A(I, J-1, K) +
          A(I+1, J, K) + A(I, J+1, K) +
          A(I, J, K-1) + A(I, J, K+1) )

// Fused Version
do KK=1,N-1
 do K=KK+1,KK,-1
  if ((K.le.N-1).and.(K.ge.2)) then
   do J=2,N-1
    do I=2+mod(KK+J+1,2),N-1,2
     A(I,J,K) = C1*A(I, J, K) + C2 *
         ( A(I-1, J, K) + A(I, J-1, K) +
           A(I+1, J, K) + A(I, J+1, K) +
           A(I, J, K-1) + A(I, J, K+1) )

// Tiled Version
do JJ=1,N-1,TJ
 do II=1,N-1,TI
  do KK=1,N-1
   do K=KK+1,KK,-1
    if ((K.le.N-1).and.(K.ge.2)) then
     do J=max(JJ+K-KK,2),
          min(JJ+K-KK+TJ-1,N-1)
      IStart=II+K-KK
      IStart=IStart+mod(KK+J+IStart+1,2)
      if (IStart.eq.1) IStart=3
      do i=IStart,min(II+K-KK+TI-1,N-1),2
       A(I,J,K) = C1*A(I, J, K) + C2 *
           ( A(I-1, J, K) + A(I, J-1, K) +
             A(I+1, J, K) + A(I, J+1, K) +
             A(I, J, K-1) + A(I, J, K+1) )
```

Figure 12 Red-black SOR examples

```
// RESID Kernel                        // Tiled RESID Kernel
                               ->   do II2=2,N-1,T2
                               ->     do II1=2,N-1,T1
do I3=2,N-1                    ->       do I3=2,N-1
  do I2=2,N-1                  ->         do I2=II2,min(N-1,II2+T2-1)
    do I1=2,N-1                ->           do I1=II1,min(N-1,II1+T1-1)
      R(I1,I2,I3)=V(I1,I2,I3)
        -A0*( U(I1,  I2,  I3  ) )
        -A1*( U(I1-1,I2,  I3  ) + U(I1+1,I2,  I3  )
                 +  U(I1,  I2-1,I3  ) + U(I1,  I2+1,I3  )
                 +  U(I1,  I2,  I3-1) + U(I1,  I2,  I3+1) )
        -A2*( U(I1-1,I2-1,I3  ) + U(I1+1,I2-1,I3  )
                 +  U(I1-1,I2+1,I3  ) + U(I1+1,I2+1,I3  )
                 +  U(I1,  I2-1,I3-1) + U(I1,  I2+1,I3-1)
                 +  U(I1,  I2-1,I3+1) + U(I1,  I2+1,I3+1)
                 +  U(I1-1,I2,  I3-1) + U(I1-1,I2,  I3+1)
                 +  U(I1+1,I2,  I3-1) + U(I1+1,I2,  I3+1) )
        -A3*( U(I1-1,I2-1,I3-1) + U(I1+1,I2-1,I3-1)
                 +  U(I1-1,I2+1,I3-1) + U(I1+1,I2+1,I3-1)
                 +  U(I1-1,I2-1,I3+1) + U(I1+1,I2-1,I3+1)
                 +  U(I1-1,I2+1,I3+1) + U(I1+1,I2+1,I3+1) )
```

Figure 13 RESID subroutine from MGRID before and after tiling

Multigrid (RESID) The last kernel is the RESID subroutine shown in Figure 13. It is a 3D stencil kernel taken from the SPECfp/NAS benchmark MGRID, a multigrid application. As an indication of its importance, MGRID is the only code in both the NAS and SPEC benchmark suites. It is also one of few SPEC 2000 codes which was preserved from SPEC95 and SPEC98.

MGRID spends 85% of its execution time performing stencil computations on large 3D arrays, with about 60% of the total execution time in RESID. The percentage increases for larger input data sets. RESID is also the kernel computation that is applied to each of the grids in the multigrid solver, though the vast majority of the time is spent on the largest (finest) grid. RESID is thus clearly an important part of MGRID.

Like JACOBI and REDBLACK, RESID also computes solutions to PDEs, but is more complex. First, instead of a 6-point stencil, RESID computes the full 27-point stencil in three dimensions. Second, RESID requires an additional input array. The fact it uses a 27-point stencil instead of a 6-point stencil means the impact of tiling is lessened, since plenty of group-reuse exists just in the first two array dimensions alone. Nonetheless, tiling improves performance, as we show later.

4.2 Methodology

Our experimental evaluation involved measuring both cache miss rates and actual program performance on a 360 MHz Sun UltraSparc2 platform. Though tiling only targeted the L1 cache, we also expect indirect improvements in L2 cache performance as shown by previous research [27]. Cache miss rates were simulated for the 16K L1 and 2M L2 direct-mapped caches present in this architecture. To compare original and optimized performance thoroughly, we varied problem sizes over a range of values when obtaining performance improvements and cache miss rate rate data. This range was selected so that the L2 cache would be able to preserve some group reuse between array planes for the smallest problem sizes, but no such group reuse for the largest problem sizes. Problem sizes $N \times N \times 30$ were used for the three kernels, where N ranged from 200 to 400. The third dimension was fixed at 30 to reduce measurement times; this had no impact on tile conflicts since conflicts in array tiles generally occur only between planes apart by 3 or less.

For each problem size, we performed each of the tiling transformations listed in Table 2, targeting the 16K L1 cache. Tile, the first tiling optimization, utilizes a fixed array tile size equal in volume to the cache size which is optimal according to the tile cost model, assuming a fully associative cache. We can therefore compare against Tile to directly examine the impact of conflict misses on the performance of tiled 3D stencils. The optimizations, Euc3D, GcdPad, and Pad were described in earlier sections. The last transformation, GcdPadNT is merely GcdPad without tiling. We consider GcdPadNT to isolate the effects of padding on performance.

	Original miss rate		Average	Transformation				
	L1	L2	Improvement	Tile	Euc3D	GcdPad	Pad	GcdPadNT
JACOBI	32.7	6.3	% perf	13	10	16	17	-1
			L1 miss rate	1.9	3.7	4.8	5.1	1.6
			L2 miss rate	0.7	0.7	0.7	0.7	-0.2
REDBLACK	22.3	4.5	% perf	89	74	120	121	10
			L1 miss rate	6.3	9.3	12.5	12.6	2.8
			L2 miss rate	2.0	1.8	2.0	2.0	-0.5
RESID	10.1	1.3	% perf	16	17	27	24	4
			L1 miss rate	1.9	2.5	4.7	4.7	2.2
			L2 miss rate	0.3	0.3	0.3	0.3	0.0

Table 3 Performance (perf) and cache miss rate improvements averaged for problem sizes 200–400 (UltraSparc2)

Program	Tile Size	Padding
Orig	(No tiling)	No
Tile	Square	No
Euc3D	Non-conflicting	No
GcdPad	Fixed non-conflicting	GCD
Pad	Variable non-conflicting	< GCD
GcdPadNT	(No tiling)	GCD

Table 2 Transformations

4.3 Overall Impact

Table 3 shows the average cache miss rates and the percent performance improvements over the range of problem sizes examined (200–400). Average miss rate improvements in this table are computed by subtracting the average miss rate from the average miss rate of the original version. Thus a drop in the average miss rate from 10 to 8 is an improvement of 2%, not 20%. Original miss rates are shown in table columns 2 and 3.

From the given average improvements in performance (*% perf*) we find that each tiling transformations leads to significant average improvements for each program. Among the tiling transformations we find that those which incorporate padding (GcdPad, Pad) perform substantially better than those that do not (Tile, Euc3D), indicating that coupling padding with tiling is essential for tiling to have the greatest impact. Average improvements for GcdPad range from 16–120% depending on the the program; improvements for Pad are nearly the same. REDBLACK, which loses spatial reuse due to non-unit stride accesses, attains the largest improvements of any kernel since tiling preserves spatial reuse in addition to temporal reuse. From GcdPadNT data we find that padding alone on average accounts for only a small part (at best) of the impact seen in GcdPad and Pad.

The average miss rate improvements indicate that tiling transformations, while targeting only the L1 cache, reduced miss rates on both caches, though much less so for the L2 cache. To explain the L2 cache miss rate improvements we note that by avoiding a cache miss on the L1 cache, we also avoid a potential miss on the L2 cache [27]. For each kernel, padding enables GcdPad and Pad to reduce roughly twice as many L1 cache misses as Tile or Euc3D. Padding alone (GcdPadNT) had a smaller impact on the L1 cache and slightly degrades L2 cache miss rates in two kernels.

4.4 Varying problem sizes

To closer examine the effectiveness of the various transformations, we give percent cache miss rates and performance in MFlops for each problem size in the range under consideration (200–400). Two figures are presented for each kernel, one figure for performance and the other for cache miss rates. Each figure consists of three graphs arranged to compare results for the various transformations.

We first examine L1 cache miss rates, which appear in Figures 14, 16, and 18 for JACOBI, REDBLACK, and RESID, respectively. We first consider JACOBI in Figure 14. From the top graph we find that while Tile and Euc3D are often able to improve L1 miss rates over Orig, their improvement is highly irregular over varying problem size. We find that the spikes in the cache miss rate for these two versions generally correspond to those problem sizes where Tile encounters significant tile conflicts or where Euc3D is forced to select a pathologically irregular tile size, as explained in Section 3.3. In contrast, GcdPad and Pad achieve overall lower and more stable L1 cache miss rates, as shown in the middle graph.

The lower graph shows that GcdPadNT is able to avoid many of the spikes in the cache miss rate which occur in the original version, preventing the loss of spatial reuse to conflict misses studied in [24]. Compared to GcdPadNT, GcdPad achieves an additional 4-5% reduction in the cache miss rate by tiling. Figures 16 and 18 show essentially the same overall relationship between the various versions on the L1 cache: Euc3D and Tile improve L1 miss rates while GcdPad and Pad achieve larger, more stable improvements.

Since L2 cache miss rates are relatively low to begin with, only modest improvements in the L2 miss rate result. These are seen mostly for the largest problem sizes, where the L2 cache lacks the capacity to preserve group reuse across different array planes. The size boundary is reached beginning at problem size 362 for JACOBI and RESID. For REDBLACK, tiling also preserves spatial reuse on the L2 cache, resulting in substantial reductions in L2 cache miss rate beginning at problem size 256.

In addition to cache miss rates, performance measurements in MFlops appear in Figures 15, 17, and 19 for JACOBI, REDBLACK, and RESID, respectively. Overall, we observe a general correlation between L1 cache miss rates and performance, as expected. While performance is highly irregular for Euc3D and Tile, it is much more stable for GcdPad and Pad.

4.5 Memory usage

We also examined the amount of additional memory used by GcdPad and Pad, the two tiling transformations which perform padding. Results for JACOBI are shown in Figure 22. Since the constant pad size upper bounds for GcdPad become less significant as problem sizes increase, we observe a general decrease in padding overhead as a percentage of the total memory size. The padding overhead for Pad is always less than that of GcdPad since Pad uses GcdPad padded array dimensions to bound its pad search. Overall, GcdPad and Pad increase the memory size by 14.7% and 4.7%, respectively. However, if we were to use the same size for the K dimension as the I and J dimensions instead of fixing it at 30 (as in actual codes), average memory size increases for GcdPad and Pad would be much less, about 1.4% and 0.5% respectively.

4.6 MGRID Benchmark

To demonstrate the potential value of our transformations for improving larger applications, we also measured overall performance improvements for MGRID after RESID is tiled. Larger applications such as MGRID can present additional challenges complicating the application of our techniques. For example, array padding cannot be performed directly in MGRID since the application indexes into three large 1D arrays to obtain the starting addresses of successive grids. Instead, we can enable padding by declaring a new padded array.

By transforming RESID using GcdPad for only the largest grid size we obtain a total execution time improvement of 6% for the reference data size ($130\times130\times130$). Although this improvement is smaller than the average improvements obtained for the RESID kernel, we expect additional improvements to arise from tiling the remaining subroutines in the application. We should note that especially large improvements are likely for applications when they utilize the pathological data sizes encountered in our varying results. In the case of MGRID, the $130\times130\times130$ problem size initially encounters a modest L1 miss rate of only 6.8%.

Experience has shown that as memory sizes and processing power increase, scientists readily exploit these advances by using larger problem sizes. For instance, the largest grid size used in MGRID has increased eightfold from SPEC95 to SPEC 2000. We consider even larger problem sizes (400–700) in Figures 20 and 21 to demonstrate the robustness of our transformations. (The overall boost in performance appears since a 450MHz UltraSparc2 was used in place of the 360MHz processor). Results indicate that our tiling techniques should remain effective even as problem sizes grow exponentially.

4.7 Discussion

Overall, our experiments show that tiling can improve performance substantially in some 3D scientific codes and that combining tile size selection and padding helps to achieve stable performance improvements. Average performance improvements for three kernels due to Pad (tiling with non-conflicting tiles and padding) range from 17–121%. Our transformations show promise for improving larger applications, especially multigrid solvers.

5 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 [33]. Computation-reordering transformations such as loop permutation and tiling are the primary optimization techniques [9, 21, 33], though loop fission (distribution) and loop fusion have also been found to be helpful [21].

Data layout optimizations such as padding and transpose have been shown to be useful in eliminating conflict misses and improving spatial locality [1, 17, 24, 25]. Data transformations have also been combined with loop transformations [5, 16].

Several cache capacity estimation techniques have been proposed to help guide data locality optimizations [9, 33]. These techniques can also be enhanced to take into account limited cache associativity [8, 30]. More recently, Ghosh *et al.* developed symbolic cache representation which are highly accurate in predicting cache misses [11, 12, 13]. Their *cache miss equations* can be used to predict the number of cache misses for a computation, and also be used to guide compiler transformations such as tiling [14].

A number of researchers have investigated tiling as a means of exploiting reuse. Tiling was first proposed by Irigoin and Triolet [15] and Wolfe [35, 36]. Lam, Rothberg, and Wolf show conflict misses can severely degrade the performance of 2D tiling [20]. Wolf and Lam analyze temporal and spatial reuse, and apply tiling when necessary to capture outer loop reuse [33],

Esseghir proposed using tall tiles consisting of the maximum number of array columns which fit in cache [7]. Coleman and McKinley select rectangular non-conflicting tile sizes [6] while others focus on using a portion of cache [34]. Temam *et al.* analyze the program to determine whether a tile should be copied to a contiguous buffer [31].

Kodukula *et al.* present a technique called data shackling which is very effective at generating tiled code for 2D complex linear algebra codes [18]. They implemented it in the SGI compiler and demonstrated its effectiveness compared to the commercial compiler [19]. Sarkar describes data locality optimizations used in the IBM XL Fortran compilers, including loop transformations and tiling [28]. Chame and Moon propose tiling algorithms for choosing tile sizes based on cost models for estimating capacity and cross-interference misses [3].

Mitchell *et al.* discussed the interactions of multi-level tiling for several goals, such as cache, TLB, and parallelism [22]. They found explicitly considering multiple levels of the memory hierarchy (cache and TLB) led to the choice of compromise tile sizes which can yield significant improvements in performance. Following their work, we found compiler optimizations targeting higher (smaller) levels of cache can frequently improve lower (larger) levels of cache as well indirectly [27].

Song and Li extending tiling techniques to handle multiple loop nests [29]. In many cases their technique can exploit reuse across multiple iterations of the time-step loop, yielding major improvements. Their technique does not currently apply to 3D arrays, and they concentrate on only L2 cache since L1 cache is too small to provided reuse for their necessarily large tiles. As explained earlier, their technique does not extend to multigrid solvers since these applications utilize a succession of smaller grid sizes.

Panda *et al.* proposed applying padding in conjunction with 2D tiling to avoid conflict misses [23]. They first pick the largest tile size which fits in cache, then select pad sizes by exhaustively testing for conflicts within the tile, incrementing pads by one whenever a conflict is found. In comparison, our algorithm is more efficient because we generate non-conflicting tile sizes directly for different pads.

Chatterjee *et al.* demonstrate that nonlinear array layouts can avoid conflict misses in tiled codes by storing data accessed in each tile using space-filling curves [4]. Alternatively, recursive divide-and-conquer algorithms can be used to obtain many of the benefits of tiling [10]. Compiler analysis can even automatically transform loop nests and generate recursive codes [38].

Finally, Weiß *et al.* explored program transformations for improving the performance numerical algorithms on hierarchical memories [32]. They applied tiling and padding transformations by hand based on domain-specific applications knowledge. We follow their work, and show how to automate these optimizations in a compiler by developing algorithms to compute pads and tile sizes.

6 Conclusions

Iterative solvers for partial differential equations (PDEs) is a key component of scientific computing. As scientists begin trying to solve problems in three dimensions, caches begin to present problems for 3D arrays. We find tiling is a program transformation compilers can apply to capture this reuse, and develop techniques to successfully apply tiling by selection of non-conflicting tiles and/or padding array dimensions to eliminate conflicts. Experiments over a range of problem sizes for key kernels and on a larger multigrid solver demonstrate the effectiveness of our transformations. We thus make progress towards our goal: helping scientists and engineers achieve high performance without having to worry excessively about machine-specific details of modern high-performance architectures.

7 Acknowledgments

We wish to thank Ulrich Rüde and Howard Elman for helpful discussions on locality issues, as well as pointing out the importance of 3D multigrid PDE solvers to the numerical analysis community.

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 hierar-

chical 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] K. Esseghir. Improving data locality for caches. Master's thesis, Dept. of Computer Science, Rice University, September 1993.

[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] K. Gatlin and L. Carter. Architecture-cognizant divide and conquer algorithms. In *Proceedings of SC'99*, Portland, OR, November 1999.

[11] 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.

[12] S. Ghosh, M. Martonosi, and S. Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, CA, October 1998.

[13] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: A compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, July 1999.

[14] S. Ghosh, M. Martonosi, and S. Malik. Automated cache optimizations using cme driven diagnosis. In *Proceedings of the 2000 ACM International Conference on Supercomputing*, Santa Fe, NM, May 2000.

[15] 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.

[16] 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.

[17] 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.

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

[19] I. Kodukula, K. Pingali, R. Cox, and D. Maydan. 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.

[20] 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.

[21] 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.

[22] 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.

[23] R. Panda, H. Nakamura, N. Dutt, and A. Nicolau. Augmenting loop tiling with data alignment for improved cache performance. *IEEE Transactions on Computers*, 48(2):142–149, February 1999.

[24] 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.

[25] G. Rivera and C.-W. Tseng. Eliminating conflict misses for high performance architectures. In *Proceedings of the 1998 ACM International Confer-*

ence on Supercomputing, Melbourne, Australia, July 1998.

[26] 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.

[27] G. Rivera and C.-W. Tseng. Locality optimizations for multi-level caches. In *Proceedings of SC'99*, Portland, OR, November 1999.

[28] 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.

[29] 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.

[30] 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.

[31] O. Temam, E. Granston, and W. Jalby. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Proceedings of Supercomputing '93*, Portland, OR, November 1993.

[32] C. Weiß, W. Karl, M. Kowarschik, and U. Rüde. Memory characteristics of iterative methods. In *Proceedings of SC'99*, Portland, OR, November 1999.

[33] 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.

[34] 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.

[35] M. J. Wolfe. Iteration space tiling for memory hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing*, December 1987.

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

[37] D. Wonnacott. Time skewing for parallel computers. In *Proceedings of the Twelfth Workshop on Languages and Compilers for Parallel Computing*, San Diego, CA, August 1999.

[38] Q. Yi, V. Adve, and K. Kennedy. Transforming loops to recursion for multi-level memory hierarchies. In *Proceedings of the SIGPLAN '00 Conference on Programming Language Design and Implementation*, Vancouver, Canada, June 2000.

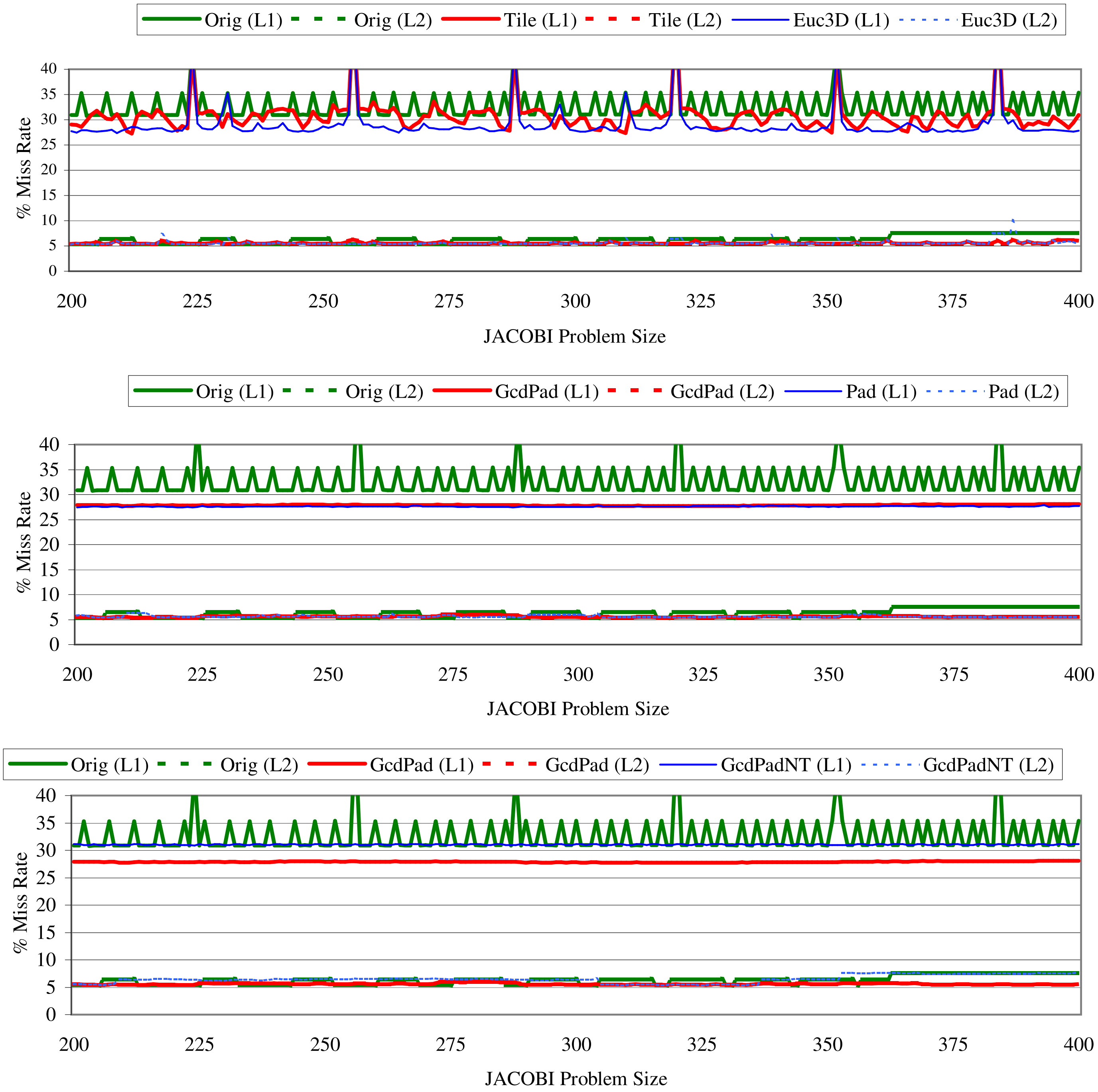

Figure 14 JACOBI: cache miss rates (16K L1, 2M L2, direct-mapped caches)

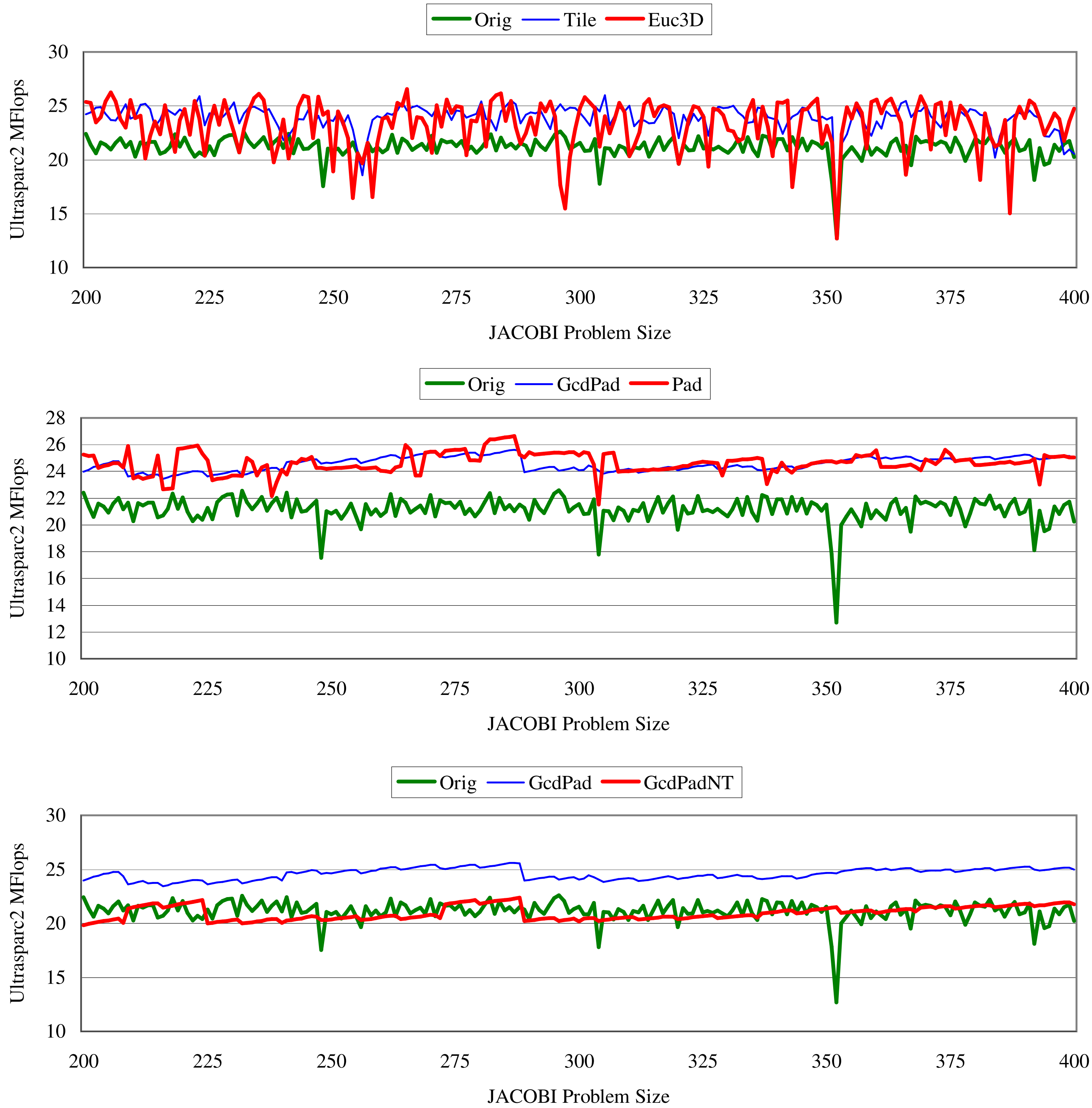

Figure 15 JACOBI: performance in MFlops on 360MHz UltraSparc2

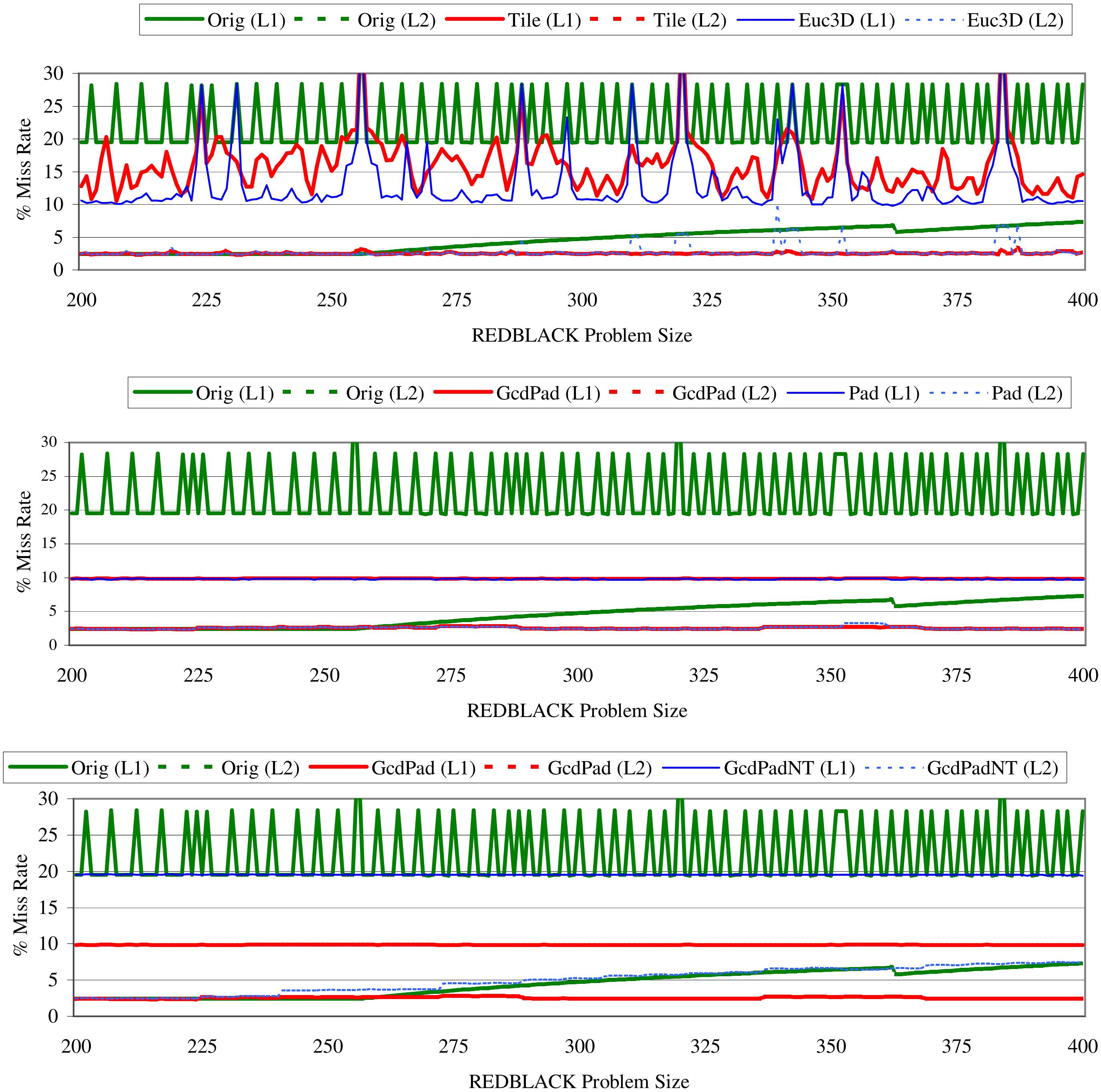

Figure 16 REDBLACK: cache miss rates (16K L1, 2M L2, direct-mapped caches)

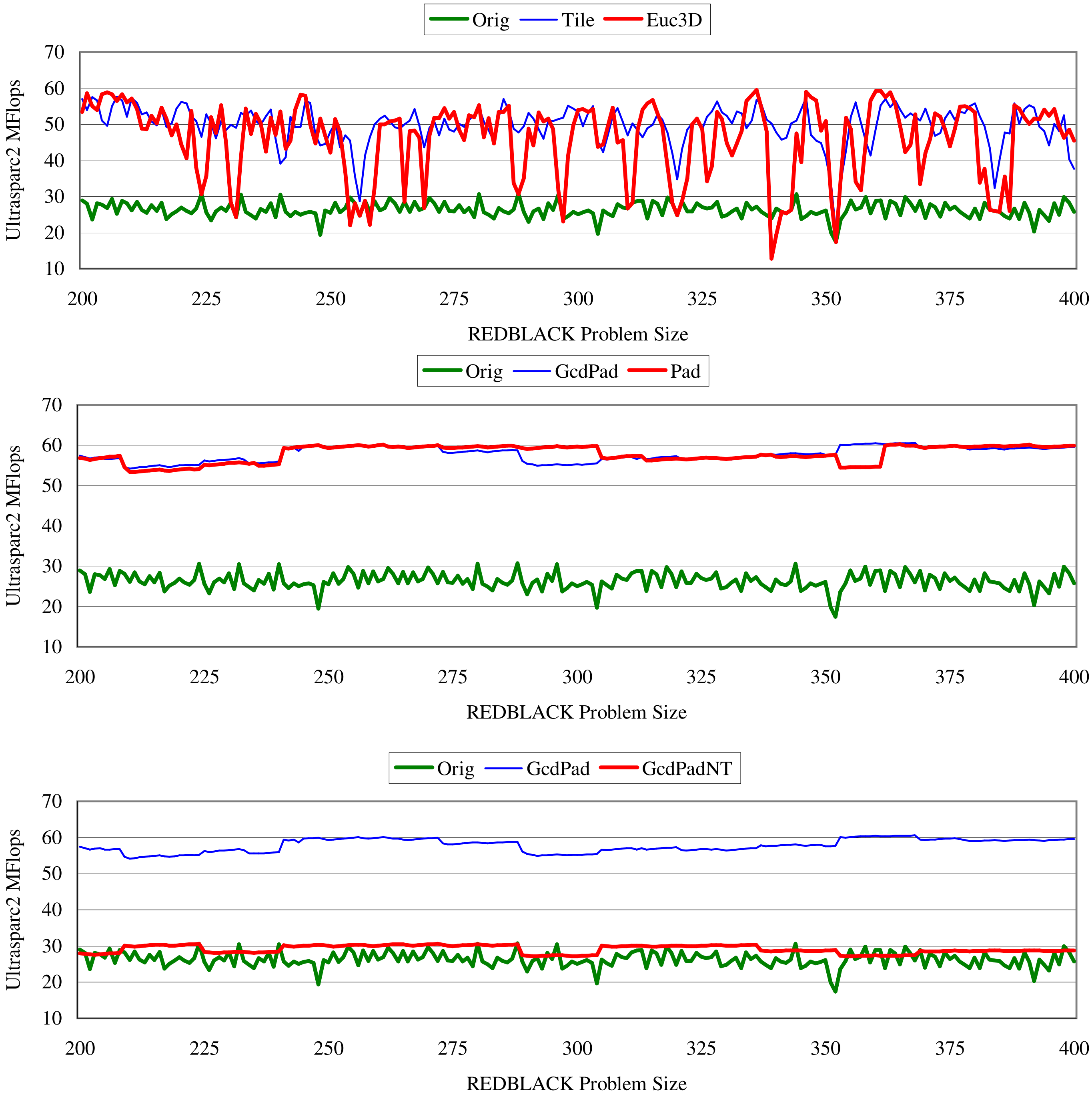

Figure 17 REDBLACK: performance in MFlops on 360MHz UltraSparc2

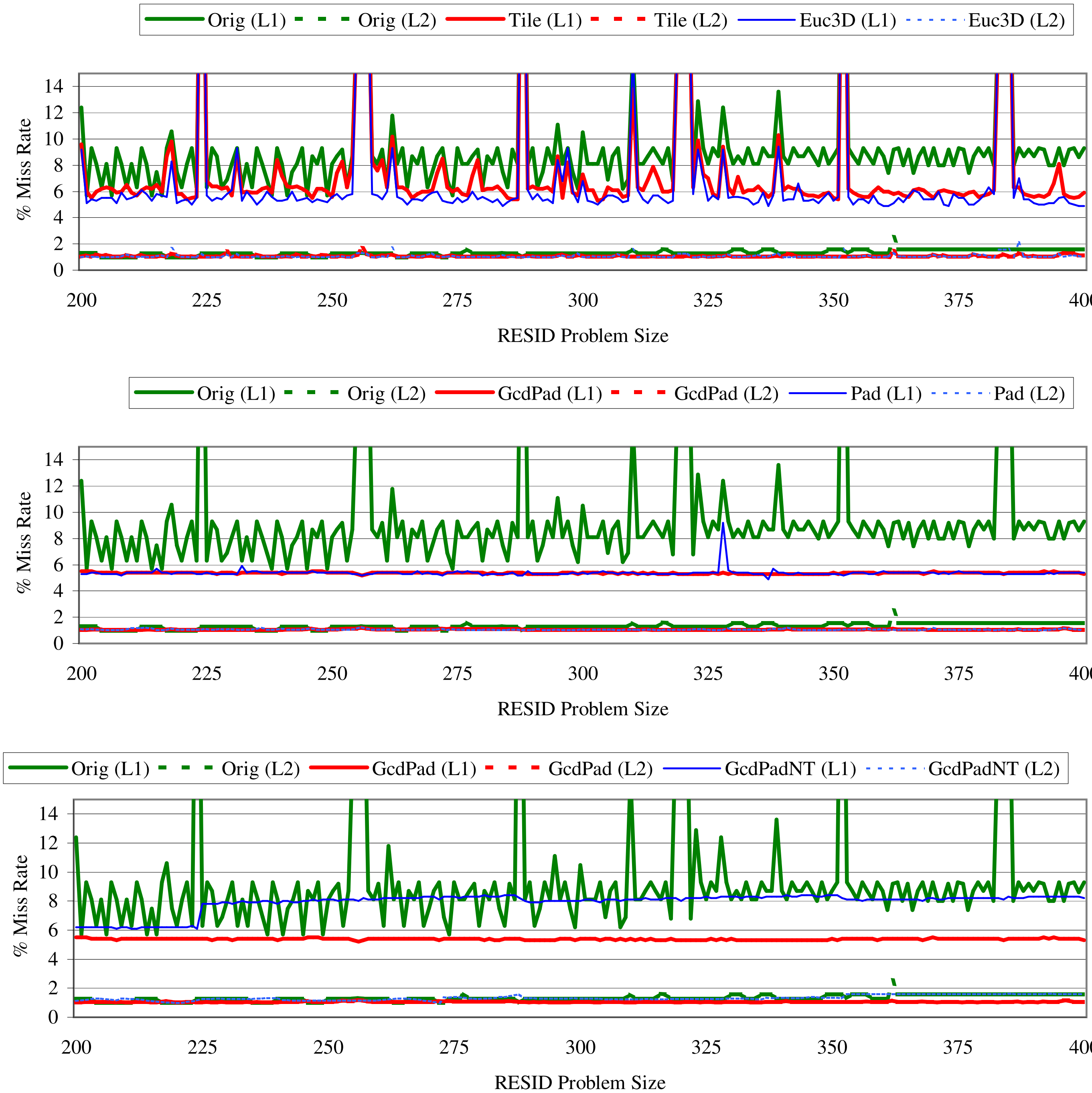

Figure 18 RESID: cache miss rates (16K L1, 2M L2, direct-mapped caches)

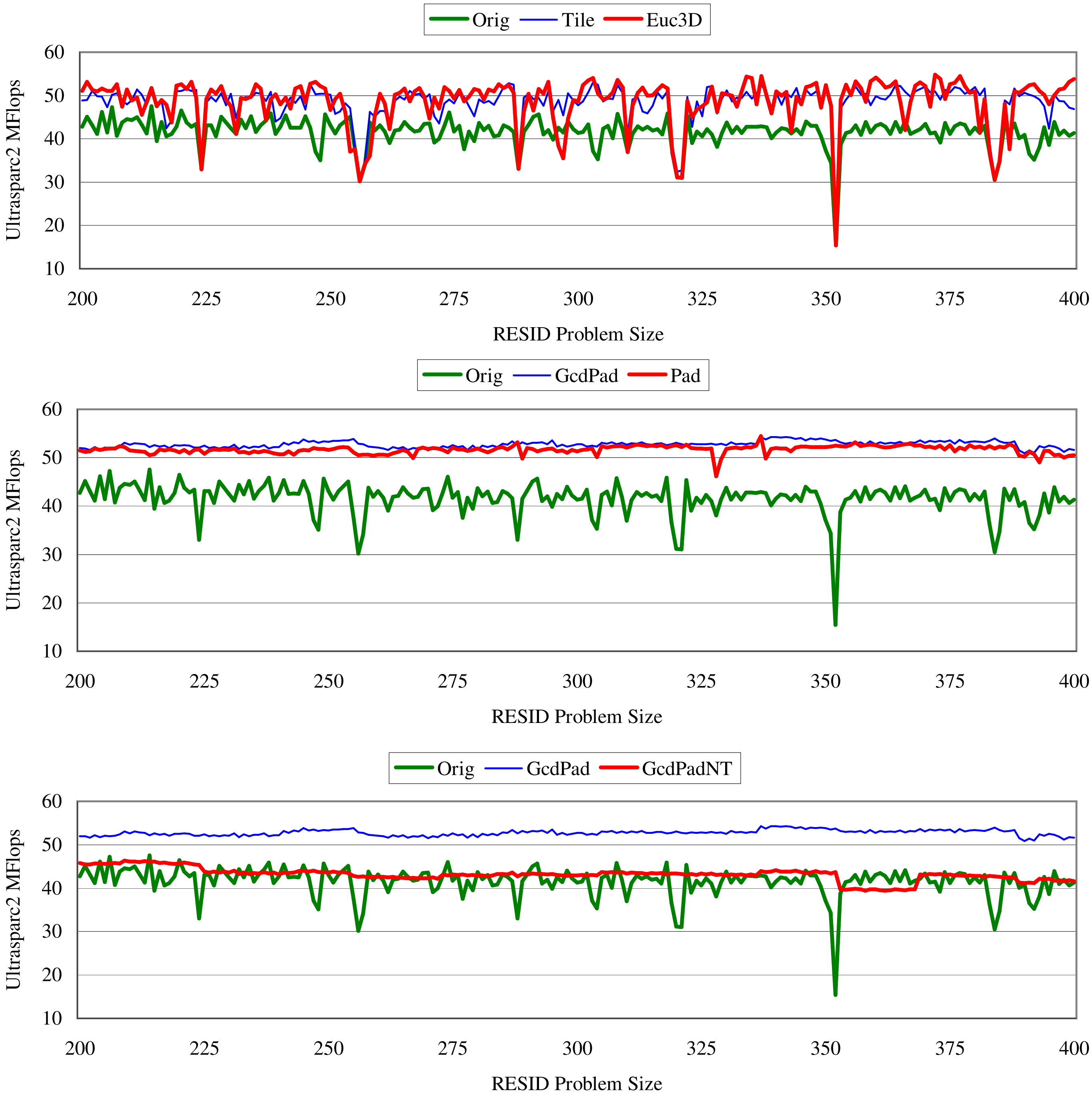

Figure 19 RESID: performance in MFlops on 360MHz UltraSparc2

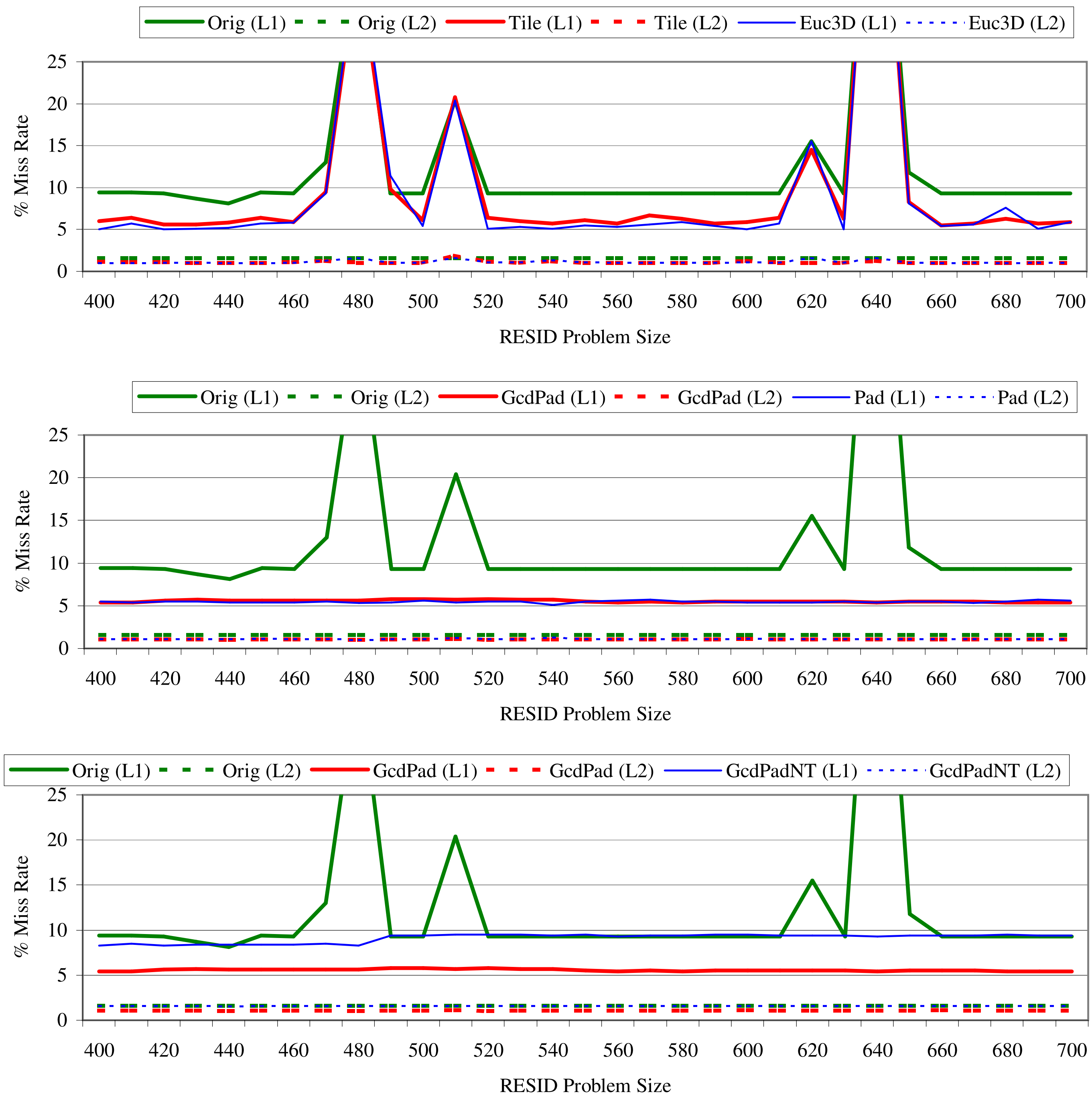

Figure 20 Larger RESID problem sizes: cache miss rates (16K L1, 2M L2, direct-mapped caches)

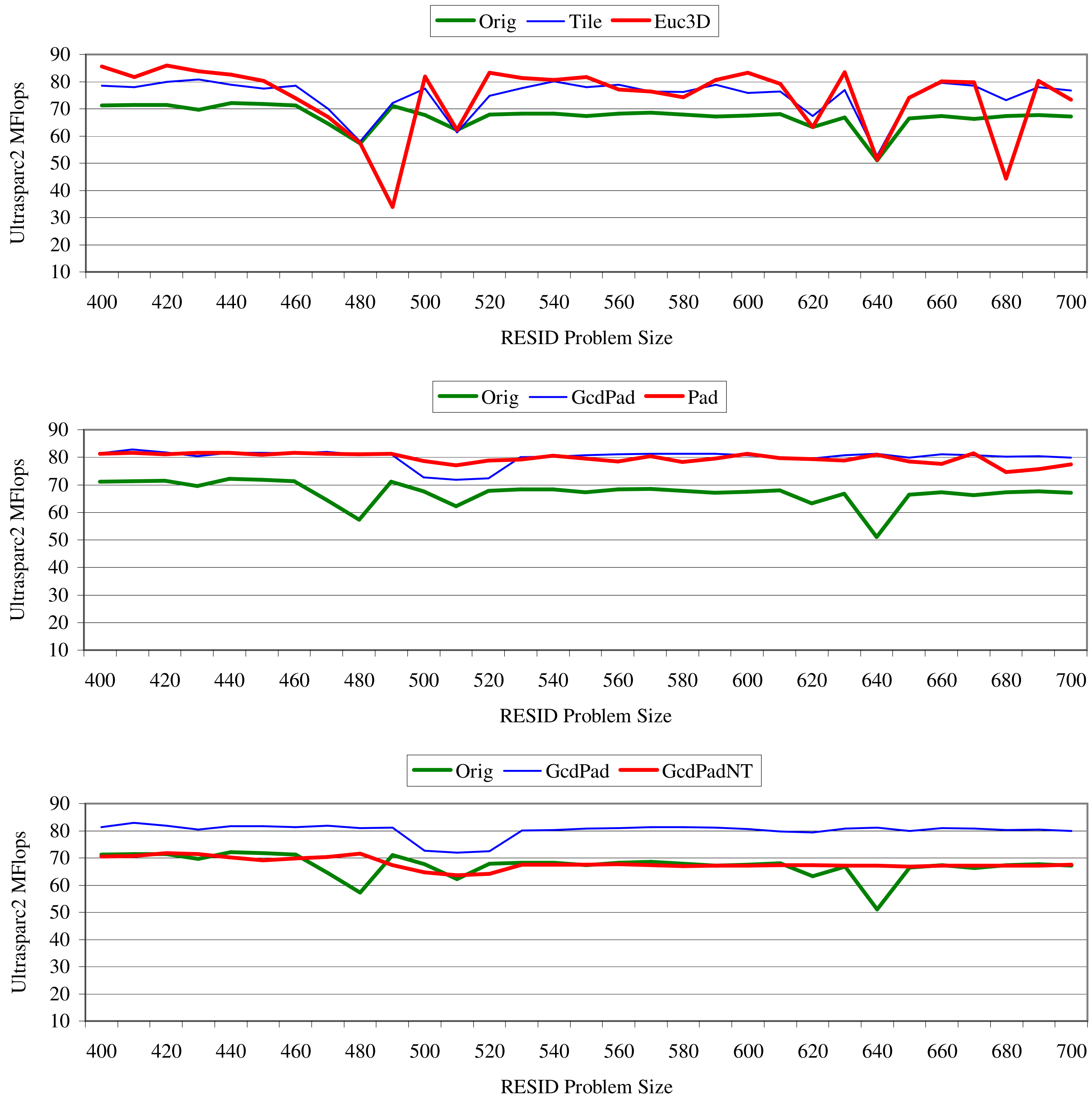

Figure 21 Larger RESID problem sizes: performance in MFlops on 450MHz UltraSparc2

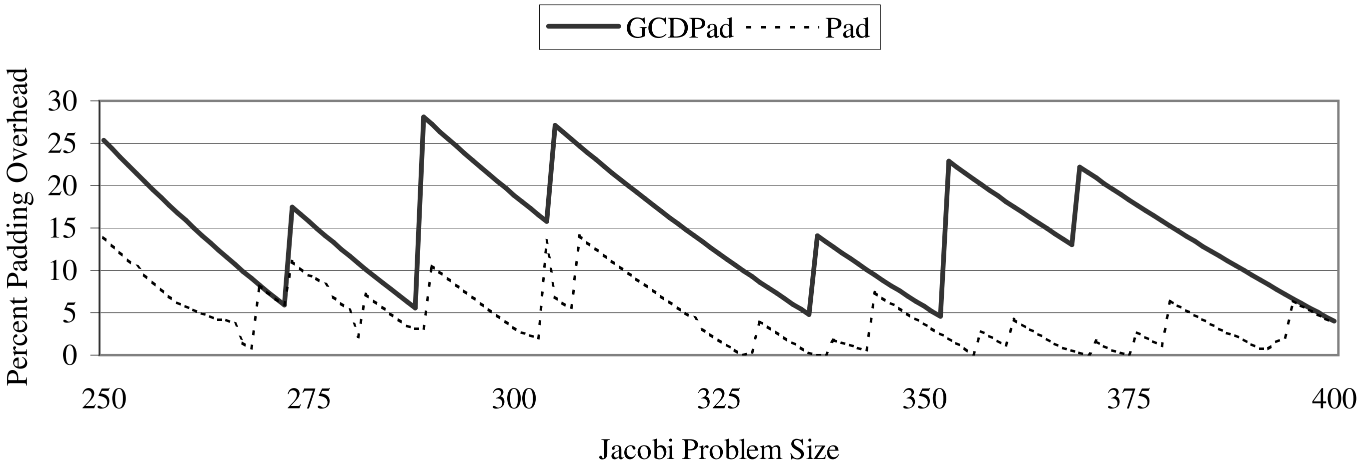

Figure 22 JACOBI: Memory increase from padding