A Comparison of Compiler Tiling Algorithms

Gabriel Rivera, Chau-Wen Tseng

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

Abstract. Linear algebra codes contain data locality which can be exploited by tiling multiple loop nests. Several approaches to tiling have been suggested for avoiding conflict misses in low associativity caches. We propose a new technique based on intra-variable padding and compare its performance with existing techniques. Results show padding improves performance of matrix multiply by over 100% in some cases over a range of matrix sizes. Comparing the efficacy of different tiling algorithms, we discover rectangular tiles are slightly more efficient than square tiles. Overall, tiling improves performance from 0-250%. Copying tiles at run time proves to be quite effective.

1 Introduction

With processor speeds increasing faster than memory speeds, memory access latencies are becoming the key bottleneck for modern microprocessors. As a result, effectively exploiting data locality by keeping data in cache is vital for achieving good performance. Linear algebra codes, in particular, contain large amounts of reuse which may be exploited through *tiling* (also known as blocking). Tiling combines strip-mining and loop permutation to create small tiles of loop iterations which may be executed together to exploit data locality [4, 11, 26]. Figure 1 illustrates a tiled version of matrix multiplication of `NxN` arrays.

```
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)
```

Fig. 1. Tiled matrix multiplication

Due to hardware constraints, caches have limited *set associativity*, where memory addresses can only be mapped to one of k locations in a k-way associative cache. *Conflict misses* may occur when too many data items map to the same set of cache locations, causing cache lines to be flushed from cache before they may be reused, despite sufficient capacity in the overall cache. Conflict

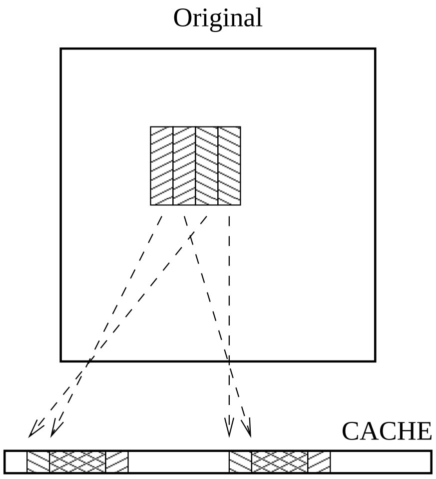

Fig. 2. Example of conflict misses and padding

misses have been shown to severely degrade the performance of tiled codes [13]. Figure 2 illustrates how the columns of a tile may overlap on the cache, preventing reuse. A number of compilation techniques have been developed to avoid conflict misses [6, 13, 23], either by carefully choosing tile sizes or by copying tiles to contiguous memory at run time. However, it is unclear which is the best approach for modern microprocessors.

We previously presented *intra-variable padding*, a compiler optimization for eliminating conflict misses by changing the size of array dimensions [20]. Unlike standard compiler transformations which restructure the computation performed by the program, padding modifies the program's *data layout*. We found intra-variable padding to be effective in eliminating conflict misses in a number of scientific computations. In this paper, we demonstrate intra-variable padding can also be useful for eliminating conflicts in tiled codes. For example, in Figure 2 padding the array column can change mappings to cache so that columns are better spaced on the cache, eliminating conflict misses.

Our contributions include:

- introducing padding to assist tiling
- new algorithm for calculating non-conflicting tile dimensions
- experimental comparisons based on matrix multiply and LU decomposition

We begin by reviewing previous algorithms for tiling, then discuss enhancements including padding. We provide experimental results and conclude with related work.

2 Background

We focus on copying tiles at run-time and carefully selecting tile sizes, two strategies studied previously for avoiding conflict misses in tiled codes. In remaining sections, we refer to the cache size as C_s, the cache line size as L_s, and the column size of an array as Col_s. The dimensions of a tile are represented by H for height and W for width. All values are in units of the array element size.

2.1 Tile Size Selection

One method for avoiding conflict misses in tiled codes is to carefully select a tile size for the given array and cache size. A number of algorithms have been proposed.

- Lam, Rothberg, and Wolf [13] pick the largest non-conflicting square tile using an $O(\sqrt{C_s})$ algorithm for selecting tile size.
- Esseghir [7] picks a rectangular tile containing the largest number of non-conflicting array columns. I.e., $H = Col_s$ and $W = \lfloor C_s/Col_s \rfloor$.
- Coleman and McKinley [6] compute rectangular non-conflicting tiles and select one using their cost model. They applied the Euclidean GCD algorithm to generate possible tile heights, where:

  $$H_{next} = H_{prev} \bmod H \tag{1}$$

  using C_s and Col_s as the initial heights. A complicated formula is presented for calculating non-conflicting tile widths, based on the gap between tile starting addresses and number of tile columns which can fit in that gap.
- Wolf, Maydan, and Chen [24] choose a square tile which uses a small fraction of the cache (5–15%) in order to avoid excessive conflicts. For instance, the tile $H = W = \lfloor \sqrt{0.10 C_s} \rfloor$ uses 10% of the cache. The particular fraction of the cache utilized is chosen based on cache characteristics such as associativity and line size.

2.2 Copying

An alternative method for avoiding conflict misses is to copy tiles to a buffer and modify code to use data directly from the buffer [13, 23]. Since data in the buffer is contiguous, self-interference is eliminated. However, performance is lost because tiles must be copied at run time. Overhead is low if tiles only need to be copied once, higher otherwise.

Figure 3 shows how copying may be introduced into tiled matrix multiply. First, each tile of `A(I,K)` may be copied into a buffer `BUF`. Because tiles are invariant with respect to the `J` loop, they only need to be copied once outside the `J` loop.

It is also possible to copy other array sections to buffers. If buffers are adjacent, then cross-interference misses are also avoided. For instance, in Figure 3

```
do KK=1,N,W
  do II=1,N,H
    copy(A(...),BUF)              // copy A(I,K)
    do J=1,N
      copy(C(...),BUF2)           // copy C(I,J)
      do K=KK,min(KK+W-1,N)
        do I=II,min(II+H-1,N)
          BUF2(...) = BUF2(...) + BUF(...) * B(K,J)
      copy(BUF2,C(...))           // copy back C(I,J)
```

Fig. 3. Tiled matrix multiplication with copying

the column accessed by array `C(I,J)` in the innermost loop is copied to `BUF2` to eliminate interference between arrays `C` and `A`. Since the location of the column varies with the `J` loop, we must copy it on each iteration of the `J` loop, causing data in `C` to be copied multiple times [23]. In addition, the data in the buffer must be written back to `C` since the copied region is both read and written to. Whether copying more array sections is profitable depends on the frequency and expense of cross-interference.

3 Tiling Improvements

We present two main improvements to existing tiling algorithms. First, we derive a more accurate method for calculating non-conflicting tile dimensions. Second, we integrate intra-variable padding with tiling to handle pathological array sizes.

3.1 Non-conflicting Tile Dimensions

We choose non-conflicting tile heights using the Euclidean GCD algorithm from Coleman and McKinley [6]. However, we compute tile widths using a simple recurrence. The recurrences for both height and width may be computed simultaneously using the recursive function *ComputeTileSizes* in Figure 4. The initial invocation is *ComputeTileSizes* $(C_s, Col_s, 0, 1)$.

ComputeTileSizes $(H, H_{next}, W_{prev}, W)$
    $H' = H - L_s + 1$ /* shrink height for long cache lines */
    /* consider tile with dimensions (H', W) */
    if $(H_{next} \geq L_s)$ then
        ComputeTileSizes $(H_{next}, H \bmod H_{next}, W, \lfloor H/H_{next} \rfloor W + W_{prev})$
    endif

Fig. 4. Recursive function for computing nonconflicting tile sizes

Table 1. H and W at invocation i given $C_s = 2048$, $Col_s = 300$

i	1	2	3	4	5	6	7
H	2048	300	248	52	40	12	4
W	1	6	7	34	41	157	512

At each invocation of *ComputeTileSizes*, a new tile size, determined by tile height H and width W, is guaranteed not to conflict when $L_s = 1$. (The proof supporting this result is too long to appear in this paper.) To account for longer cache lines, an adjusted tile height $H' = H - L_s + 1$ is used in place of H. By subtracting most of the cache line size L_s from the tile height H, we slightly under-utilize the cache but guarantee no conflicts will occur. To choose between the different non-conflicting tile dimensions, we select the tile (H, W) minimizing $\frac{1}{H} + \frac{1}{W}$. This cost function favors square tiles over rectangular tiles with the same area; it is similar to that used by Coleman and McKinley [6].

Table 1 illustrates the sequence of H and W values computed by *ComputeTileSizes* at each invocation when $C_s = 2048$ and $Col_s = 300$. An important result is that each of the computed tile sizes are *maximal* in the sense that neither their heights nor widths may be increased without causing conflicts. Moreover, *ComputeTileSizes* computes *all* maximal tile sizes. Note that at invocation 1, (H, W) is not a legal tile size since $H = 2048$ exceeds Col_s. In general, this can occur only at the first invocation, and a simple comparison with Col_s will prevent consideration of such tile sizes. The formula used by *ComputeTileSizes* for finding non-conflicting tile widths is simpler than that of the Coleman and McKinley algorithm. In addition, it avoids occasionally incorrect W values that result from their algorithm.

3.2 Padding

Our second improvement is to incorporate intra-variable padding with tiling. Previously we found memory access patterns common in linear algebra computations may lead to frequent conflict misses for certain pathological column sizes, particularly when we need to keep two columns in cache or prevent self-interference in rows [20]. Bailey [2] first noticed this effect and defined *stride efficiency* as a measure of how well strided accesses (e.g., row accesses) avoid conflicts. Empirically, we determined that these conflicts can be avoided through a small amount of intra-variable padding. In tiled codes a related problem arises, since we need to keep multiple tile columns/rows in cache.

Table 2. H and W at invocation i given $C_s = 2048$, $Col_s = 768$

i	1	2	3	4
H	2048	768	512	256
W	1	2	3	8

Table 3. Tiling Heuristics

Program version	Description
orig	No tiling
ess	Largest number of non-conflicting columns (Esseghir)
lrw	Largest non-conflicting square (Lam, Rothberg, Wolf)
tss	Maximal non-conflicting rectangle (Coleman, McKinley)
euc	Maximal (accurate) non-conflicting rectangle (Rivera, Tseng)
wmc10	Square tile using 10% of cache (Wolf, Maydan, Chen)
lrwPad	**lrw** with padding
tssPad	**tss** with padding
eucPad	**euc** with padding
eucPrePad	**euc** with pre-copying to padded array
copyTile	Tiles of array `A` copied to contiguous buffer
copyTileCol	Tiles of array `A` and column of `C` copied to contiguous buffer

When *ComputeTileSizes* obtains tile sizes for pathological column sizes, though the resulting tile sizes are nonconflicting, overly "skinny" or "fat" (nonsquare) tiles result, which decrease the effectiveness of tiling. For example, if $Col_s = 768$ and $C_s = 2048$, *ComputeTileSizes* finds only the tile sizes shown in Table 2. The tile closest to a square is still much taller than it is wide. For this Col_s, any tile wider than 8 will cause conflicts. This situation is illustrated Figure 2, in which the column size for the array on the left would result in interference with a tile as tall as shown. On the right we see how padding enables using better tile sizes. Our padding extension is thus to consider pads of 0–8 elements, generating tile sizes by running *ComputeTileSizes* once for each padded column size. The column size with the best tile according to the cost model is selected. By substituting different cost models and tile size selection algorithms, we may also combine this padding method with the algorithms used by Lam, Rothberg, Wolf and Coleman and McKinley.

Padding may even be applied in cases where changing column sizes is not possible. For example, arrays passed to subroutines cannot be padded without interprocedural analysis, since it is not known whether such arrays require preserving their storage order. In many linear algebra codes the cost of pre-copying to padded arrays is often small compared to the cost of the actual computation. For instance, initially copying all of `A` to a padded array before executing the loop in Figure 1 adds only $O(N^2)$ operations to an $O(N^3)$ computation. We may therefore combine padding with tile size selection by either directly padding columns or by pre-copying.

4 Experimental Evaluation

4.1 Evaluation Framework

To compare tiling heuristics we varied the matrix sizes for matrix multiplication (MULT) from 100 to 400 and applied the heuristics described in Table 3. For each

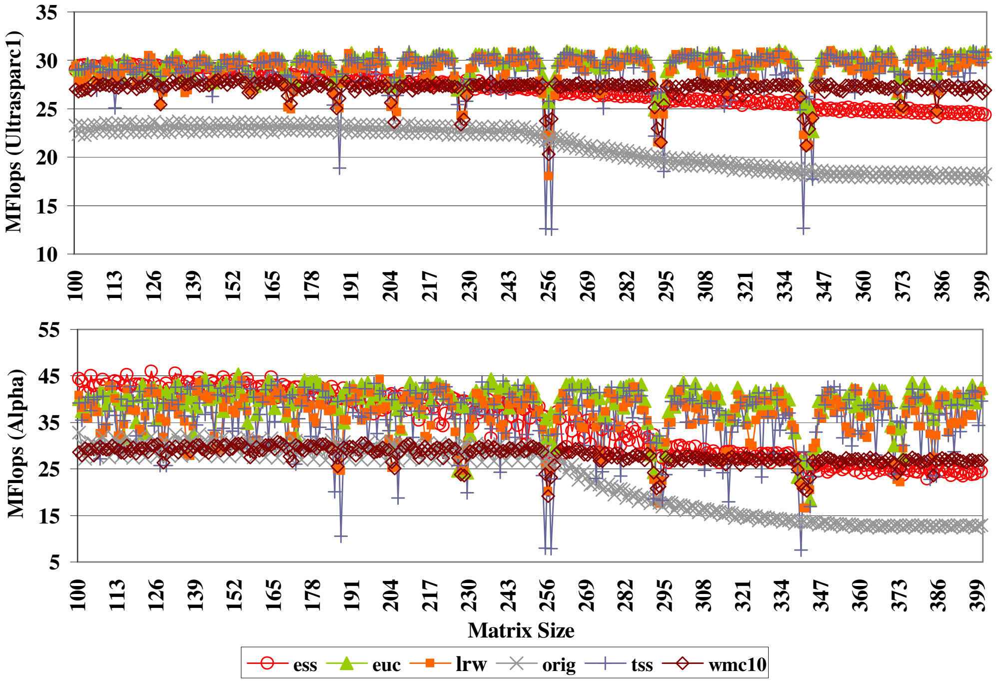

Fig. 5. Matrix multiplication: MFlops of tiling heuristics

heuristic, performance on a Sun UltraSparc I and a DEC Alpha 21064 were measured. Both processors use a 16k direct-mapped Level 1 (L1) cache. In addition, several heuristics were applied to varying problem sizes of LU decomposition (LU). We also computed the percent cache utilization for several heuristics.

4.2 Performance of MULT

Tile Size Selection. We first consider heuristics which do not perform copying or padding. Ultra and Alpha megaflop rates of MULT for these heuristics are graphed in Figure 5. The X-axis represents matrix size and the Y-axis gives Mflops. In the top graph we see that tiled versions usually outperform **orig** versions by 4 or more Mflops on the Ultra, improving performance by at least 20%. We find that for sizes beginning around 200, **ess** and **wmc10**, the heuristics which do not attempt maximality of tile dimensions, obtain a lesser order improvement than **euc**, **lrw**, and **tss**, usually by a margin of at least 2 Mflops. Performance of the latter three heuristics appears quite similar, except at the clusters of matrix sizes in which performance of all heuristics drops sharply. In these cases we see **euc** does not do nearly as bad, and that **tss** drops the most. The lower graph gives the same data with respect to the Alpha. Behavior is similar, though variation in performance for individual heuristic increases, and **ess** is a competitive heuristic until matrix sizes exceed 250.

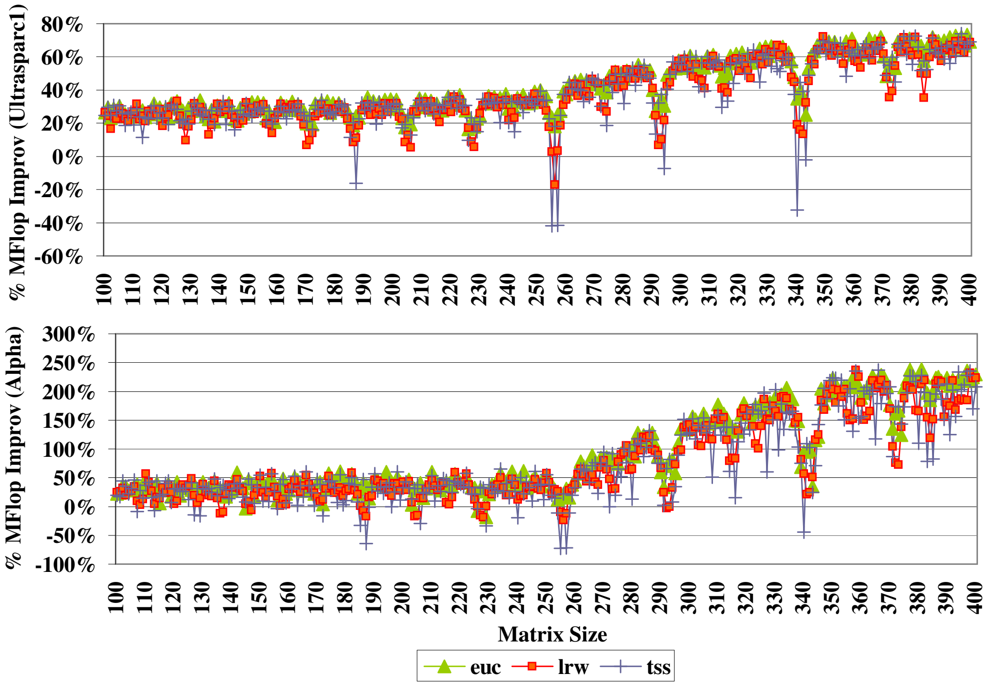

Fig. 6. Matrix multiplication: MFlop improvements of tiling heuristics

These results illuminate the limitations of several heuristics. Lower **ess** performance indicates a cost model should determine tile heights instead of the array column size, as using the column size results in overly "skinny" tiles. Lower **wmc10** performance underscores the need to better utilize the cache. **tss** would benefit from accuracy in computing tile widths. A prominent feature of both graphs is the gradual dip in performance of **orig** and **ess** beginning at 256. This occurs as matrix sizes exceed the Level 2 (L2) cache, indicating **ess** is also less effective in keeping data in the L2 cache than other heuristics.

The top graph in Figure 6 focuses on **euc**, **lrw**, and **tss**, giving percent Mflops improvements on the Ultra compared to **orig**. While all heuristics usually improve performance by about 25%–70%, we again observe clusters of matrix sizes in which performance drops sharply, occasionally resulting in degradations (negative improvement). **euc** does best in these cases, while **lrw** and especially **tss** do considerably worse. The lower graph shows results are similar on the Alpha, but the sudden drops in performance tend to be greater. Also, performance improvements are much larger beyond 256, indicating L2 cache misses are more costly on the Alpha.

Averaging over all problem sizes, **euc**, **lrw**, and **tss** improve performance on the Ultra by 42.1%, 38.0%, and 38.5%, respectively, and by 92.3%, 76.6%, and 76.4% on the Alpha. The advantage of **euc** over **lrw** indicates that using only square tiles is an unfavorable restriction. For instance, at problem size 256,

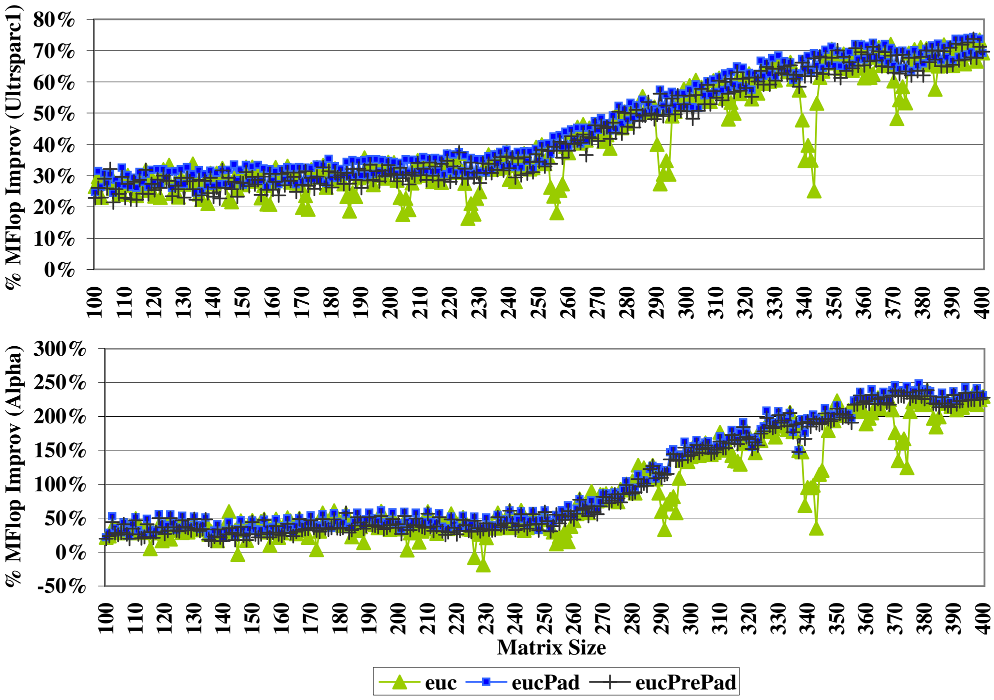

Fig. 7. Matrix multiplication: MFlop improvements of tiling w/ padding

where **euc** selects a 256x8 tile, **lrw** selects an 8x8 tile, at the expense of over 40% of performance on both architectures. Though **euc** handles such problem sizes better than **lrw**, performance still degrades for **euc** since the only tile sizes possible at this column size are too "skinny". Thus, pathological problem sizes adversely affect all three heuristics dramatically.

Padding. To avoid pathological problem sizes which hurt performance, we combine padding with tile size selection. Figure 7 compares **euc** with **eucPad** and **eucPrePad**. In both graphs, **eucPad** and **eucPrePad** improvements demonstrate that padding is successful in avoiding these cases. Moreover, the cost of pre-copying is acceptably small, with **eucPrePad** attaining improvements of 43.3% on the Ultra whereas **eucPad** improves performance by 45.5%. On the Alpha, **eucPrePad** averages 98.5% whereas **eucPad** averages 104.2%. Since pre-copying requires only $O(N^2)$ instructions, the overhead becomes even less significant for problem sizes larger than 400. Improvements for **lrwPad** and **tssPad**, which do not appear in Figure 7, resemble those of **eucPad**. Both are slightly less effective, however. On average, **lrwPad** and **tssPad** improve performance on the Ultra by 44.3% and 43.8% respectively.

Copying Tiles. An alternative to padding is to copy tiles to a contiguous buffer. Figure 8 compares improvements from **copyTile** and **copyTileCol** with those of **eucPad**, the most effective noncopying heuristic. On the Ultra, **copyTile** is as stable as **eucPad**, and overall does slightly better, attaining an average

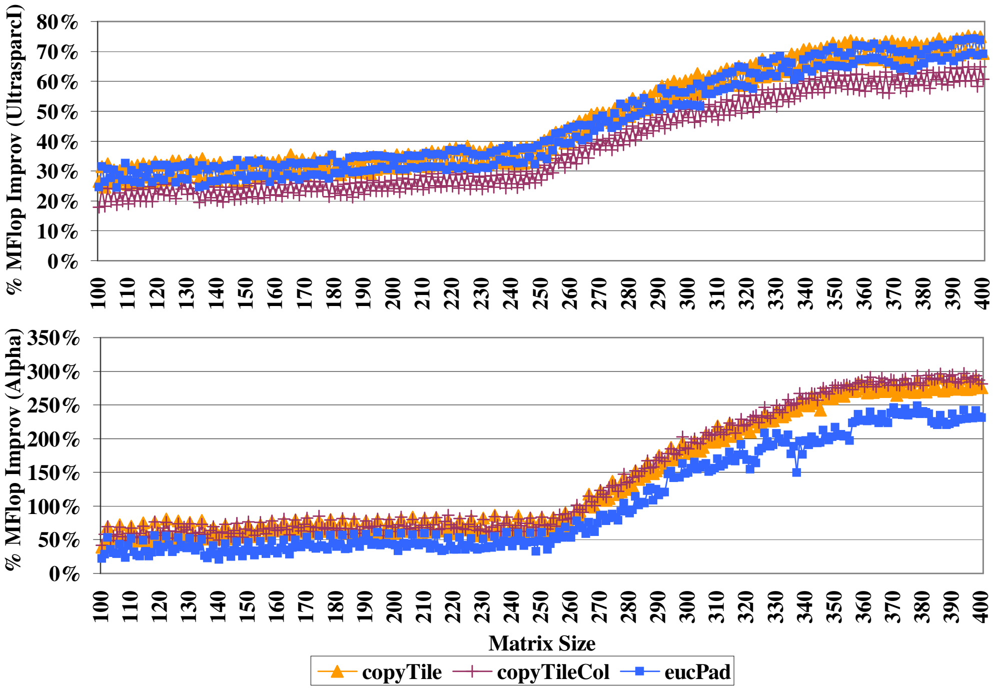

Fig. 8. Matrix multiplication: MFlop improvements of copying heuristics

improvement of 46.6%. Though **copyTileCol** is just as stable, overhead results in improvements consistently worse than both **eucPad** and **copyTile**, and the average improvement is only 38.1%. We find a different outcome on the Alpha, on which both **copyTile** and **copyTileCol** are superior to **eucPad**. This is especially true for larger matrix sizes, where copying overhead is less significant.

Summary. From the above results, we observe that tile size selection heuristics which compute maximal square or rectangular non-conflicting tiles are most effective. Also, padding can enable these heuristics to avoid pathological cases in which substantial performance drops are unavoidable. Moreover, we find copying tiles to be advantageous in MULT.

4.3 Performance of LU

Tile Size Selection. We also compare padding heuristics **euc** and **lrw** on LU. Figure 9 gives percent Mflops improvements for **euc** and **lrw**. As with MULT, on both the Ultra and the Alpha, large drops in performance occur at certain clusters of matrix sizes, and **euc** is again more effective in these cases. However, tiling overhead has a greater impact, leading to frequent degradations in performance until tiling improves both L1 and L2 cache performance at 256. As a result, overall improvements on the Ultra for **euc** and **lrw** are only 17.8% and 11.4%, respectively. On the Alpha, overall performance, even worse for matrix sizes less than 256, is 53.8% and 31.6% for **euc** and **lrw**.

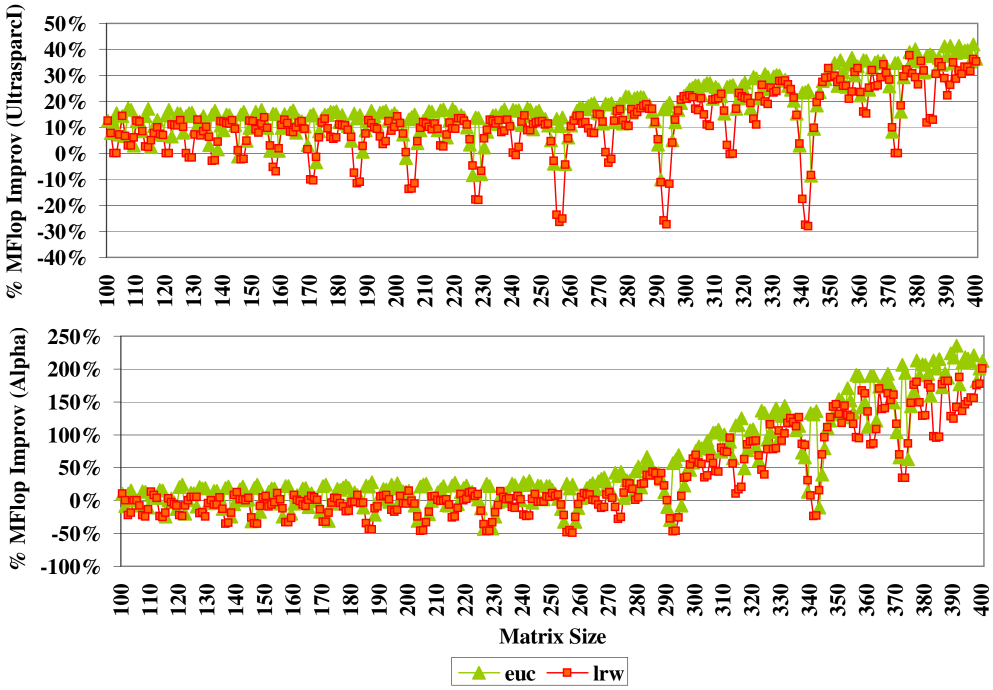

Fig. 9. LU Decomposition: MFlop improvements of tiling heuristics

Padding. We see again that padding helps to stabilize performance in Figure 10. In the top graph, **eucPad** and **eucPrePad** consistently improve Ultra Mflop rates, achieving overall improvements of 19.7% and 15.5% respectively. An interesting feature of this graph are the three spikes in performance of **eucPad** and **eucPrePad** at 128, 256, and 384. These correspond to column sizes containing a large power-of-2 factor, leading to ping-pong conflict misses between references to unpadded arrays [20]. Thus, a side effect of padding to prevent tile self-interference is the elimination of ping-pong cross-interference misses in some cases. The lower graph shows that padding stabilizes performance improvements on the Alpha as well, but large performance increases do not begin until 256. Average improvements for **eucPad** and **eucPrePad** are 58.5% and 49.8% respectively.

4.4 Cache Utilization

Finally, cache utilization, computed as HW/C_s, appears in Figure 11 for four heuristics. The top graph give cache utilization for **euc** and **lrw**. Here, the X-axis again gives problem size while the Y-axis gives percent utilization for a 16k cache. We see that for **lrw**, which chooses only square tiles, utilization varies dramatically for different matrix sizes. Low cache utilization for **lrw** occurs when the largest nonconflicting square tile is very small. For matrix size 256, for instance, **lrw** computes an 8x8 tile. Utilization for **euc** is comparatively high, since it may choose rectangular tiles. The lower graph gives cache utilization

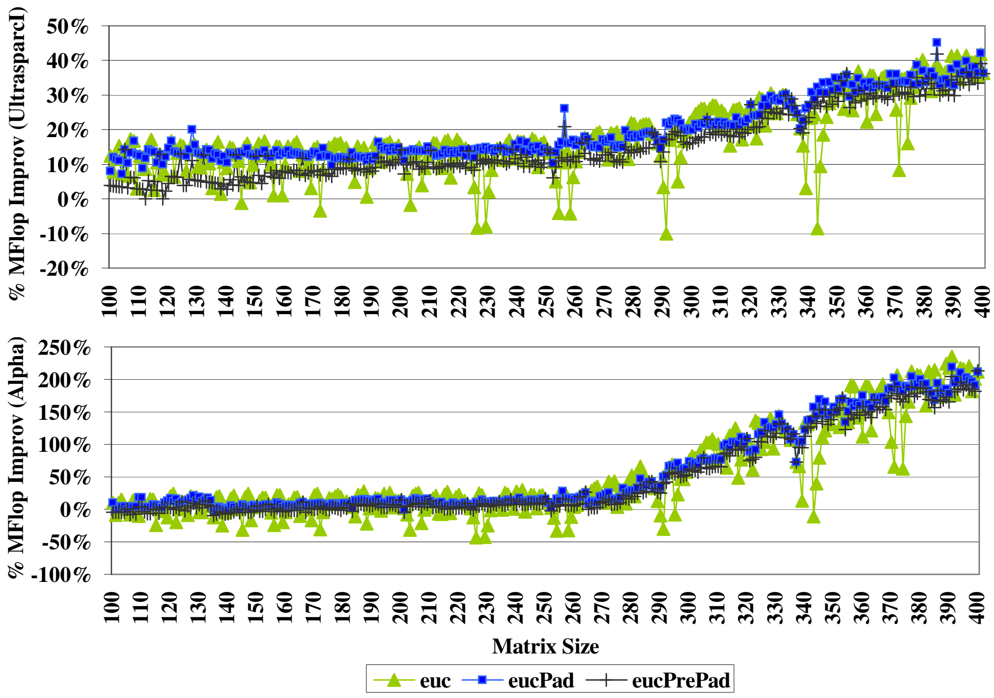

Fig. 10. LU Decomposition: MFlop improvements of tiling w/ padding

for **eucPad** and **lrwPad**. Utilization for **lrwPad** is much higher overall than for **lrw** since padding is used to avoid small tiles. Often we see utilization by both **lrwPad** and **eucPad** remain level for small intervals of problem sizes. This occurs when an especially favorable tile is available at a particular column size N. In these cases, **lrwPad** and **eucPad** will perform the padding necessary to attain that tile in several of the problem sizes leading up to N.

5 Related Work

Data locality has been recognized as a significant performance issue for both scalar and parallel architectures. A number of researchers have investigated tiling as a means of exploiting reuse. Lam, Rothberg, Wolf show conflict misses can severely degrade the performance of tiling [13]. Wolf and Lam analyze temporal and spatial reuse, and apply tiling when necessary to capture outer loop reuse [25], Coleman and McKinley select rectangular non-conflicting tile sizes [6] while others focus on using a portion of cache [24]. Temam *et al.* analyze the program to determine whether a tile should be copied to a contiguous buffer. Mitchel *et al.* study interactions between tiling for multiple objectives at once [16].

In addition to tiling, researchers working on locality optimizations have considered both computation-reordering transformations such as loop permuta-

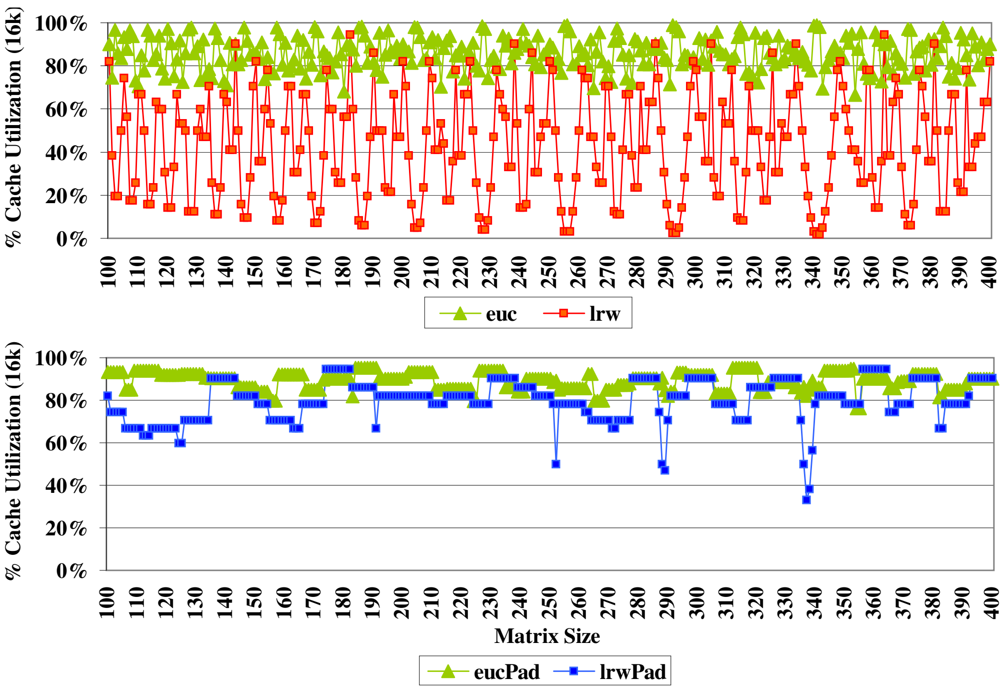

Fig. 11. Matrix multiplication: Cache utilization of tiling heuristics

tion [9, 17, 25] and loop fission/fusion [15, 17]. Scalar replacement replaces array references with scalars, reducing the number of memory references if the compiler later puts the scalar in a register [3]. Many cache models have been designed for estimating cache misses to help guide data locality optimizations [8, 9, 17, 25]. Earlier models assumed fully-associative caches, but more recent techniques take limited associativity into account [10, 22].

Researchers began reexamining conflict misses after a study showed conflict misses can cause half of all cache misses and most intra-nest misses in scientific codes [18]. Data-layout transformations such as array transpose and padding have been shown to reduce conflict misses in the SPEC benchmarks when applied by hand [14]. Array transpose applied with loop permutation can improve parallelism and locality [5, 12, 19]. Array padding can also help eliminate conflict misses [1, 20, 21] when performed carefully.

6 Conclusions

The goal of compiler optimizations for data locality is to enable users to gain good performance without having to become experts in computer architecture. Tiling is a transformation which can be very powerful, but requires fairly good knowledge of the caches present in today's advanced microprocessors. In this paper, we have examined and improved a number of tiling heuristics. We show non-conflicting tile widths can be calculated using a simple recurrence, then demonstrate intra-variable padding can avoid problem spots in tiling. Experimental results on two architectures indicate large performance improvements are possible using compiler heuristics. By improving compiler techniques for automatic tiling, we allow users to obtain good performance without considering machine details. Scientists and engineers will benefit because it will be easier for them to take advantage of high performance computing.

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. D. Bailey. Unfavorable strides in cache memory systems. Technical Report RNR-92-015, NASA Ames Research Center, May 1992.
3. D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, NY, June 1990.
4. S. Carr and K. Kennedy. Compiler blockability of numerical algorithms. In *Proceedings of Supercomputing '92*, Minneapolis, MN, November 1992.
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. 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, J. Ramanujam, and A. Choudhary. A compiler algorithm for optimizing locality in loop nests. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, Vienna, Austria, July 1997.

13. 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.
14. A. Lebeck and D. Wood. Cache profiling and the SPEC benchmarks: A case study. *IEEE Computer*, 27(10):15–26, October 1994.
15. N. Manjikian and T. Abdelrahman. Fusion of loops for parallelism and locality. *IEEE Transactions on Parallel and Distributed Systems*, 8(2):193–209, February 1997.
16. N. Mitchell, L. Carter, J. Ferrante, and K. Hogstedt. 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.
17. K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.
18. K. S. McKinley and O. Temam. A quantitative analysis of loop nest locality. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, Boston, MA, October 1996.
19. M. O'Boyle and P. Knijnenburg. Non-singular data transformations: Definition, validity, and applications. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, Vienna, Austria, July 1997.
20. G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.
21. 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.
22. 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.
23. 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.
24. M. 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.
25. 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.
26. M. J. Wolfe. More iteration space tiling. In *Proceedings of Supercomputing '89*, Reno, NV, November 1989.

This article was processed using the LaTeX macro package with LLNCS style