Augmenting Loop Tiling with Data Alignment for Improved Cache Performance

Preeti Ranjan Panda, *Member*, *IEEE*, Hiroshi Nakamura, *Member*, *IEEE*, Nikil D. Dutt, *Senior Member*, *IEEE*, and Alexandru Nicolau, *Member*, *IEEE*

Abstract—Loop blocking (tiling) is a well-known compiler optimization that helps improve cache performance by dividing the loop iteration space into smaller blocks (tiles); reuse of array elements within each tile is maximized by ensuring that the working set for the tile fits into the data cache. Padding is a data alignment technique that involves the insertion of dummy elements into a data structure for improving cache performance. In this work, we present DAT, a technique that augments loop tiling with data alignment, achieving improved efficiency (by ensuring that the cache is never under-utilized) as well as improved flexibility (by eliminating self-interference cache conflicts independent of the tile size). This results in a more stable and better cache performance than existing approaches, in addition to maximizing cache utilization, eliminating self-interference, and minimizing cross-interference conflicts. Further, while all previous efforts are targetted at programs characterized by the reuse of a single array, we also address the issue of minimizing conflict misses when several tiled arrays are involved. To validate our technique, we ran extensive experiments using both simulations as well as actual measurements on SUN Sparc5 and Sparc10 workstations. The results on benchmarks exhibiting varying memory access patterns demonstrate the effectiveness of our technique through consistently high hit ratios and improved performance across varying problem sizes.

Index Terms—Loop tiling, data cache, data alignment, cache conflict.

1 Introduction

The growing disparity between processor and memory speeds makes the efficient utilization of cache memory a critical factor in determining program performance. Compiler optimizations to exploit the data cache have been studied and implemented in the past. Compulsory cache misses, which occur when memory data is referenced for the first time, can be minimized to some extent by prefetching techniques [1]. Reduction of capacity misses (which occur when reusable data is evicted from the cache due to cache size limitations) is achieved by *loop blocking* (or *loop tiling*) [2]—a well-known compiler optimization that helps improve cache performance by dividing the loop iteration space into smaller *blocks* (or *tiles*). This also results in a logical division of arrays into tiles. The *working set* for a tile is the set of all elements accessed during the computation involving the tile. Reuse of array elements within each tile is maximized by ensuring that the working set for the tile fits into the data cache. Conflict misses, which occur when several data elements compete for the same cache location, are a consequence of limited-associativity caches. Cache conflicts in the context of tiled loops are categorized into two classes: 1) *self-interference*: cache conflicts among array elements in the same tile, and 2) *cross-interference*: cache conflicts among elements of different tiles (of the same array) or different arrays. Conflict misses can seriously degrade program performance; their reduction is generally addressed during the selection of tile sizes.

Data alignment techniques for reducing cache misses were reported by Lebeck and Wood [3] in a study of the cache performance of the SPEC92 benchmark suite, where they observed significant speedups (up to 3.4X) even on code that was previously tuned using execution-time profilers.

Padding is a data alignment technique that involves the insertion of dummy elements in a data structure for improving cache performance. In this paper, we present DAT, a data alignment technique based on padding of arrays that ensures a more stable and, in most cases, better cache performance than existing approaches, in addition to maximizing cache utilization, eliminating self-interference, and minimizing cross-interference. Further, while all previous efforts are targeted at programs characterized by the reuse of a single array, we also address the issue of minimizing conflict misses when several tiled arrays are involved. Our experiments on several benchmarks with varying memory access patterns demonstrate the effectiveness of our technique and its stability with respect to changing problem sizes.

The rest of the paper is organized as follows. In Section 2, we discuss previous published work on loop tiling for cache performance enhancement. In Section 3, we demonstrate the significance of the padding technique through a motivating example. In Section 4, we describe our tile size selection and padding strategies and a generalization of the strategies into the case with several tiled arrays. We describe our experimental results in Section 5, discuss the implications in Section 6, and conclude in Section 7.

- *P.R. Panda is with Synopsys, Inc., 700 E. Middlefield Rd., Mountain View, CA 94043. E-mail: panda@synopsys.com.*
- *H. Nakamura is with the University of Tokyo, Research Center for Advanced Science and Technology, 4-6-1 Komaba, Meguro-ku, Tokyo 153-8904, Japan. E-mail: nakamura@hal.rcast.u-tokyo.ac.jp.*
- *N.D. Dutt and A. Nicolau are with the Department of Information and Computer Science, University of California at Irvine, Irvine, CA 92697. E-mail: {dutt, nicolau}@ics.uci.edu.*

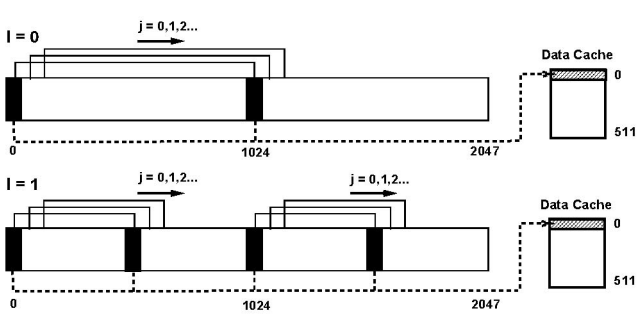
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEE CS Log Number 108226.

```
Algorithm FFT_Original
Input: n, wreal[n * 2^(n-1)],
  wimag[n * 2^(n-1)]
Inout: sigreal[2^n], sigimag[2^n]
le = 2^n; windex = 1; wptrind = 0;
for (l = 0; l < n; l++) {
  le = le/2;
  for (j=0; j < le; j++) {
    wpr = wreal [wptrind];
    wpi = wimag [wptrind];
    for (i=j; i < 2^n; i += 2*le) {
      xureal = sigreal [i];
      xuimag = sigimag [i];
      xlreal = sigreal [i + le];
      xlimag = sigimag [i + le];
      sigreal [i] =
        xureal + xlreal;
      sigimag [i] =
        xuimag + xlimag;
      tr = xureal - xlreal;
      ti = xuimag - xlimag;
      sigreal [i + le] =
        tr * wpr - ti * wpi;
      sigimag [i + le] =
        tr * wpi - ti * wpr;
    } /* loop i */
    wptrind += windex;
  } /* loop j */
  windex = windex * 2;
} /* loop l */
```

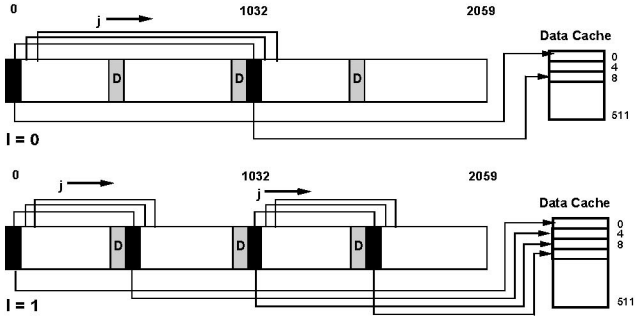
(a)

```
Algorithm FFT_Padded
Input: n, wreal[n * 2^(n-1)],
  wimag[n * 2^(n-1)],
  CACHE_SZ, LINE_SZ
Constant: NO_LINES =
  CACHE_SZ / LINE_SZ
  NEW_SZ = 2^n + 2^n/NO_LINES
Inout: sigreal[NEW_SZ],
  sigimag[NEW_SZ]
le = 2^n; windex = 1; wptrind = 0;
for (l = 0; l < n; l++) {
  le = le/2;
  le' = le + le / NO_LINES;
  for (j=0; j < le; j++) {
    wpr = wreal [wptrind];
    wpi = wimag [wptrind];
    for (i=j; i < 2^n; i += 2*le) {
      i' = i + i / NO_LINES;
      xureal = sigreal [i'];
      xuimag = sigimag [i'];
      xlreal = sigreal [i' + le'];
      xlimag = sigimag [i' + le'];
      sigreal [i'] =
        xureal + xlreal;
      sigimag [i'] =
        xuimag + xlimag;
      tr = xureal - xlreal;
      ti = xuimag - xlimag;
      sigreal [i' + le'] =
        tr * wpr - ti * wpi;
      sigimag [i' + le'] =
        tr * wpi - ti * wpr;
    } /* loop i */
    wptrind += windex;
  } /* loop j */
  windex = windex * 2;
} /* loop l */
```

(b)



(c)



(d)

Fig. 1. (a) Original FFT algorithm, (b) modified FFT algorithm, (c) distribution of array accesses in data cache for original FFT algorithm, (d) distribution of array accesses in data cache for modified FFT algorithm.

2 Previous Work

Several techniques for exploiting the data cache through loop tiling have been proposed in the past. Loop restructuring techniques to enable tiling are reported in [4], [2], [5], all of which do not address conflict misses that occur in real caches.

Lam et al. [6] reported the first work modeling interferences in direct-mapped caches with a study of the cache performance of a matrix multiplication program for different tile sizes. They present an algorithm (LRW) for computing the tile size which selects the largest square tile that does not incur self-interference conflicts. This strategy

is effective for reducing cache misses, but is sensitive to the array size—the tile sizes vary widely with small changes in array sizes. For instance, in the matrix multiplication example, for a 1,024-element cache, a 200×200 array results in a 24×24 tile, whereas a 205×205 array results in a 5×5 tile. The 5×5 tile makes inefficient utilization of the 1,024-element cache and degrades performance due to loop overheads introduced by tiling.

Esseghir [7] presents a tile size selection algorithm (ESS) for one-dimensional tiling which selects a tile with as many rows of the array, as would fit into the data cache. This algorithm cannot exploit the benefits of two-dimensional tiling and also does not consider cross-interference among arrays in its computation of tile sizes.

Coleman and McKinley [8] present a technique (TSS) based on the Euclidean G.C.D. computation algorithm for selecting a tile size that attempts to maximize the cache utilization while eliminating self-interferences within the tile. They incorporate the effects of the cache line size, as well as cross-interference between arrays. The tile sizes generated by TSS are, like LRW, also very sensitive to the array dimension. For the matrix multiplication example on a 1,024-element cache, a 200×200 array results in a 41×24 tile, whereas, a 205×205 array results in a 4×203 tile. Since the working set in both cases is large, the cache utilization of TSS is good. However, if the working set gets too close to the cache size, cross-interferences, which are handled with a probabilistic estimate in TSS, begin to degrade the performance. We discuss this issue in Section 5.

In this work, we present a data alignment technique based on padding of arrays that ensures a stable cache performance for a variety of problem sizes. Our approach differs from that presented in [6], [7], [8] in that we are also able to handle the case where several tiled arrays are involved. As in previous work, we focus our attention on two-dimensional arrays.

3 Motivating Example

We illustrate the effect of the padding data alignment technique on the cache performance of the Fast Fourier Transform (FFT) algorithm. Fig. 1a shows the core loop of FFT [9], highlighting the accesses to array *sigreal*.

Fig. 1c illustrates the distribution of the accesses in the *sigreal* array for two iterations of the outer loop (indexed by *l*) with array size $n = 2,048$ words, and an example 512-word direct-mapped cache with four words per line. In the first iteration of the outer loop ($l = 0$), we have accesses to the set of pairs $(sigreal[i], sigreal[i + 1,024])$, all of which conflict in the cache because, in general, the pairs $(sigreal[i], sigreal[i + 512k])$ will map to the same cache location for all integral k. Similarly, we observe similar severe conflicts between the pairs $(sigreal[i], sigreal[i + 512])$ in the second iteration ($l = 1$).

The pathological conflicts above can be prevented by the judicious padding of the *sigreal* array with dummy elements. Fig. 1d shows a modification of the storage of the *sigreal* array with four dummy words (size of one cache line) inserted after every 512 words. This ensures that $sigreal[i]$ and $\mathrm{sigreal[i + 2^n]}$ never map into the same cache line. The *sigimag* array, which has an identical access pattern to *sigreal*, is prevented from conflicting with *sigreal* by adjusting the distance between the two arrays, i.e., by inserting padding between them.

A comparison of the execution times of the original FFT algorithm in Fig. 1a with that of Fig. 1b shows a speedup of 15 percent on the SunSparc-5 machine. This shows that the time spent in the extra computation involved in calculating the new array indices (i' and le') is small in comparison to the time saved by preventing the cache misses.

4 Data Alignment Strategy

We now describe our technique, DAT, for data alignment of two-dimensional arrays. In Section 4.1, we describe the procedure for selecting the tile sizes. For the selected tile sizes, we compute the required padding of arrays for avoiding self-interference within the tile in Section 4.2. In Section 4.3, we present a generalization of the technique to handle the case when a tiled loop involves several arrays with closely-related access patterns.

4.1 Tile Size Computation

The tile size selection procedure can be summarized as follows: Select the largest tile for which the working set fits into the cache. For computing the working set size, we use the formulation used in [8].

Fig. 2 describes procedure *SelectTile* for computing the tile sizes. In addition to the most common case where we select the square tile, we also allow the user to specify the *shape* of the tile in cases where additional information about the application is available. To achieve this, the procedure takes the parameters: *TypeX, TypeY, ShapeX,* and *ShapeY*. In the common case (case 1: *ShapeX* = *ShapeY* = 0), we choose the largest square tile for which the working set fits into the cache. $ws(x, y)$ gives the size of the working set for a tile with x rows and y columns. We maintain the number of columns to be a multiple of the cache line size to ensure that unnecessary data is not brought into the cache. If *ShapeX* and *ShapeY* have a nonzero value, the user wishes to guide the tile-selection process. *TypeX* and *TypeY* can take the value CONSTANT or VARIABLE. If *TypeX* (*TypeY*) has value CONSTANT and *TypeY* (*TypeX*) has value VARIABLE, as in cases 2 and 3, the user has fixed the number of rows (columns) in the tile as *ShapeX* (*ShapeY*). Procedure *SelectTile* only determines the number of columns (rows). If both *TypeX* and *TypeY* are CONSTANT (case 4), the user has already optimized the tile sizes, and the procedure performs no action. If both *TypeX* and *TypeY* are VARIABLE (case 5), the user intends the ratio of rows and columns of the tile to be *ShapeX* : *ShapeY*. The procedure selects the largest tile with the rows and columns in the given ratio for which the working set $\leq C$.

4.1.1 Associative Caches

Associative caches can be effectively utilized in our tiling procedure to reduce cross interference in a tiled loop. We first determine the *largest allowed working set size* (M) for the tile by considering the possibility of cross-interference. If there is no cross-interference (e.g., there is only one array involved), we set $M = C$ (the cache size). However, there are circumstances where cross-interferences cannot be

completely eliminated. For example, in the matrix multiplication example (discussed in Section 5), implementing $P_1 \times P_2 = P_3$, where P_1, P_2, and P_3 are matrices, the tiling algorithm is designed to exploit reuse in matrix P_2—the other matrices contribute unavoidable cross-interferences with the tile from P_2 in the cache. In this case, if the cache associativity $A > 1$, we set $M = \frac{A-1}{A} \times C$. [1] In other words, in an A-way cache, in any cache entry (with A lines), the *primary* tile (tile of the blocked array) occupies only $(A-1)$ lines, leaving the remaining one line to be occupied by elements outside the reused tile. This ensures that elements of the primary tile are almost never mistakenly evicted from the cache. After determining M, we use procedure *SelectTile* (Fig. 2), replacing the condition $ws(x, y) \leq C$ by $ws(x, y) \leq M$.

4.2 Pad Size Computation

The computation of tile size in Section 4.1 ignores the possibility of self-interferences within the tile. We now formulate the procedure for an appropriate padding of the rows of the array with dummy elements in order to avoid this self-interference. Consider the mapping of a 30 rows $\times$ 30 columns tile in a 256×256 array into a $1,024$-element cache. Fig. 3a shows that rows 1 and 5 of the tile cause self-interference because they map into the same cache locations. To overcome this conflict, we can pad each row of the array with eight dummy elements so that the fifth tile row now occupies the space adjacent to row 1 in the cache (Fig. 3b). The regularity of the cache mapping ensures that no self-interference occurs among the elements of all the tile rows.

Algorithm *ComputePad* (Fig. 4) outlines the pad size computation procedure. The initial assignment to *InitPad* ensures that the rows of the padded array are aligned to the cache line size. For different multiples of *LineSize*, from a minimum of 0 to a maximum of *CacheSize*, we test if the resulting padded row of size $(N + PadSize)$ causes self-interference conflicts for the given tile of dimension *Rows* $\times$ *Cols*. This is done by iterating through all the elements of the tile in steps of *LineSize* and checking if any two elements map into the same cache line.

4.3 Multiple Tiled Arrays

The padding technique described in Sections 4.1 and 4.2 lends itself to an elegant generalization when computing the tile sizes of an algorithm involving more than one tiled array. We assume that the arrays have identical sizes $(S \times R)$. Since both arrays are accessed in the same tiled loop, they have identical tile sizes. We choose the tile sizes such that, as before, the working set (which involves elements accessed from all the tiled arrays) fits into the cache. To determine the padding size such that both self-interferences within each tile, as well as cross-interferences across tiles, are avoided, we first construct a tile consisting of the smallest rectangular shape (T) enclosing all the tiles. In Fig. 5, the new contour formed from the tiles in arrays A and B are shown. We now determine the pad size such that rectangle T in an $S \times R$ array does not have any self-interference, using the method described in Section 4.2. This becomes the pad size of all the tiled arrays. However, the tile size of each array remains $X \times Y$. Finally, we adjust the distance between the starting points of the arrays so that the tiles are laid out the way they appear in rectangle T in the first iteration. For example, in Fig. 5, arrays A and B (both are now $S \times R'$) are laid out such that $A[0][0]$ and $B[0][0]$ are a distance $((X \cdot R') \bmod C)$ apart in the cache.

5 Experiments

We performed experiments on two commonly available workstations—SUNSparc5 (SS5) and SUNSparc10 (SS10). The SS5 has a direct-mapped 8 KB data cache (1,024 double precision elements) with a 16-byte cache line (two elements per line). The SS10 has a 4-way associative 16 KB data cache (2,048 elements) with a 32-byte cache line (four elements per line). Our experimental results were performed both through actual measurement (MFLOPs on SUN SPARC Stations) as well as simulations on the SHADE simulator from SUN Microsystems [10].

The example programs on which we performed our experiments are: 1) Matrix Multiplication (MM) (with the standard *ijk*-loop nest permuted to *ikj*-order, as in [6]); 2) Successive Over Relaxation (SOR) [8]; 3) L-U Decomposition (LUD) [8]; and 4) Laplace [11]. We did not use the FFT algorithm in the comparisons because it does not involve tiled loops and, consequently, LRW, TSS, and ESS cannot be applied to it.

5.1 Uniformity of Cache Performance

Our first experiment was to verify our claim that the padding technique results in a uniformly good performance for a wide variety of array sizes. Fig. 6 shows the variation of data cache miss ratios (on the SS5 cache configuration) of four algorithms (LRW, ESS, TSS, and DAT) on the matrix multiplication (MM) example for all integral array sizes between 35 and 350.[2] We observe that the miss ratio of DAT is consistently low, independent of problem size, whereas all other algorithms show some sensitivity to the size. This is attributed to the fact that DAT uses fixed tile dimensions (30×30) for the given cache parameters, independent of array size, whereas, in the other algorithms, tile dimensions vary widely with array size.

5.2 Variation of Cache Performance with Problem Size

We present experiments below on the performance of each technique on the various examples, for several array sizes. We include array sizes of 256, 300, 301, and 550 to enable comparison with TSS [8] (which also presents data on these sizes). Array sizes 256 and 512 are chosen to illustrate the case with pathological cache interference, while 300 and 301 are chosen to demonstrate the widely different tile sizes for small changes in array size. Sizes 384, 640, and 768 illustrate the cases where the array size is a small multiple of a power

1. This expression for M is somewhat arbitrary. If, for each tile of T elements, there is a maximum of K cross-interfering elements, the expression M = (T/(K + T))*C is also a possibility.

2. For small array sizes, all the data fits into the cache, obviating the necessity for tiling; for larger sizes, the simulation times on the commercial simulator, SHADE, were too long to examine every integral data size. Hence, the range 35-350. We expect similar performance for larger arrays.

Procedure *SelectTile*
Input:
TypeX, *TypeY*, *ShapeX*, *ShapeY*,
Cache Size (C), Associativity (A), Cache Line Size (L)
Output:
SizeX, *SizeY* /* No. of rows and columns of selected tile */

Case 1 ($ShapeX = ShapeY = 0$):
/* Common case: no application-specific info. available */
$SizeX = SizeY = \mathbf{max}\{k|(k = L \cdot t, t \in I) \wedge (ws(k,k) \leq C)\}$
Case 2 ($TypeX$ = CONSTANT **and** $TypeY$ = VARIABLE):
/* Fixed number of rows */
$SizeX = ShapeX$
$SizeY = \mathbf{max}\{k|(k = L \cdot t, t \in I) \wedge (ws(SizeX,k) \leq C)\}$
Case 3 ($TypeY$ = CONSTANT and $TypeX$ = VARIABLE):
/* Fixed number of columns */
$SizeY = ShapeY$
$SizeX = \mathbf{max}\{k|(k \in I) \wedge (ws(k,SizeY) \leq C)\}$
Case 4 ($TypeX$ = CONSTANT and $TypeY$ = CONSTANT):
/* Tile size already determined */
$SizeX = ShapeX; SizeY = ShapeY$
Case 5 ($TypeX$ = VARIABLE and $TypeY$ = VARIABLE):
/* $ShapeX/ShapeY$ gives ratio of #rows and #cols in tile */
$(SizeX, SizeY) = (ShapeX \cdot t, ShapeY \cdot t)$ such that
$t = \mathbf{max}\{k|(k \in I) \wedge (ws(ShapeX \cdot k, ShapeY \cdot k) \leq C)\}$

Fig. 2. Procedure for selecting tile size.

of 2 ($384 = 128 \times 3; 640 = 128 \times 5; 768 = 256 \times 3$). We also include data for array sizes 700, 800, 900, and 1,000.

Fig. 7 shows the cache performance of the blocked L-U Decomposition program described in [8]. An analysis of the array access patterns in LUD [12] reveals the following relationship for the optimal tile shape: No. of Columns = No. of Rows × Cache Line Size.

Fig. 7 shows that DAT and LRW have the lowest cache **miss ratios** (except at array sizes such as 256, 384, 512, etc., where LRW has considerably higher miss ratios). The lower miss ratio of LRW is, however, often at the expense of instruction count, since the relatively smaller tiles in LRW incur overheads introduced by loop tiling. TSS tends to incur higher miss ratios because it sometimes chooses a comparatively larger tile sizes, leading to the working set size being too close to the cache size, which results in cross-interferences. ESS incurs cache conflicts because it does not account for cross-interferences.

A comparison of the **instruction count** shows that, in general, ESS has a smaller number of executed instructions than DAT and LRW. This is a consequence of the larger tile sizes selected by ESS. The instruction count of TSS fluctuates because it selects widely varying tile sizes.

The processor **cycle count** (with memory latency = 12 cycles; normalized to DAT) and **MFLOPs** measurements show that DAT has the best and most stable overall performance.

Table 1 summarizes the comparisons of experimental results for all the four examples (MM, LUD, SOR, and Laplace) on the SS5 and SS10 platforms on the basis of the four metrics identified above. In order to do a fair comparison, the table does not include array sizes such as 512, 640, etc., which would penalize all the other techniques. Thus, the improvements shown in the table are obtained without considering these cases. If these were taken into account, the improvement would be higher. Column 1

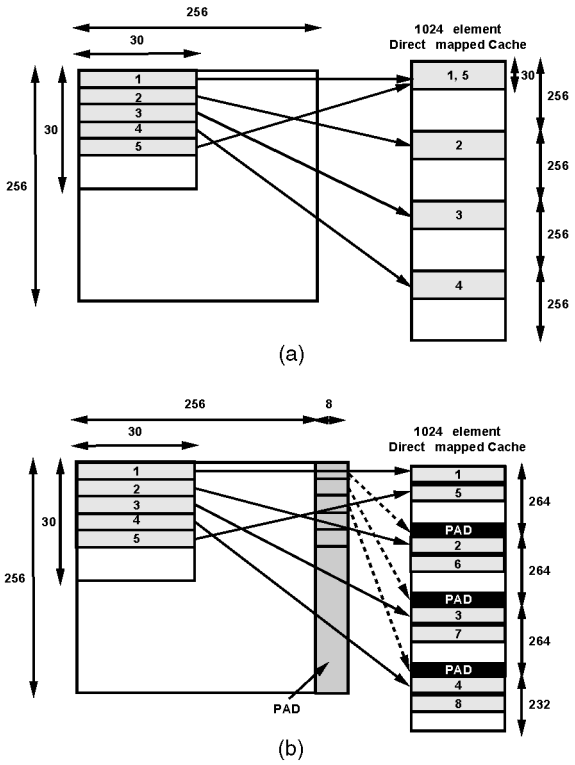


Fig. 3. (a) Self-interference in a 30×30 tile: Rows 1 and 5 of tile interfere in cache. (b) Avoiding self-interference with padding.

gives the example names; Columns 2, 3, and 4 give the improvement in cache miss ratios (MR) of our technique DAT over TSS, ESS, and LRW, respectively. Similarly, Columns 5, 6, and 7 compare the normalized instruction counts; Columns 8, 9, and 10 compare the normalized processor cycle counts; and Columns 11, 12, and 13 compare the MFLOPs with respect to DAT, i.e., the speedup

Algorithm *ComputePad*
Input:
Array Dimensions: $N' \times N$; /* Row Size = N */
Tile Dimensions: $Rows \times Cols$
Cache Size (in words): $CacheSize$
Cache Line Size (in words): $LineSize$
Output:
Pad Size for Array: $PadSize$

if ($N \bmod LineSize == 0$) **then** $InitPad = 0$
else $InitPad = LineSize - N \bmod LineSize$
for $PadSize = InitPad$ **to** $CacheSize$ **step** $LineSize$
$Status$ = OK
Initialize $LinesArray[i] = 0$ for all i
$(0 \leq i \leq \lceil CacheSize/LineSize \rceil)$
for $i = 0$ **to** $Rows$
for $j = 0$ **to** $Cols$ **step** $Linesize$
$k = (i \times (N + PadSize) + j) \bmod CacheSize$
if ($LinesArray[k/$LineSize$] == 1$)
then $Status$ = CONFLICT
else $LinesArray[k/$LineSize$] = 1$
end for /* loop j */
end for /* loop i *
if ($Status ==$ OK) **then return** $PadSize$
end for /* loop $PadSize$ */

Fig. 4. Algorithm for computing pad size for array.

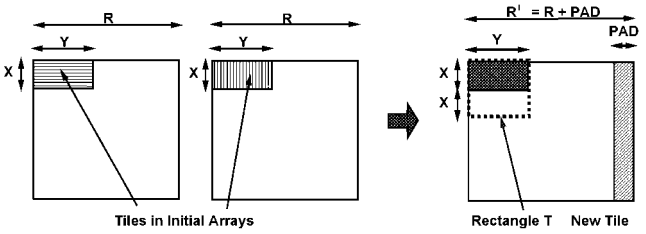


Fig. 5. Multiple tiled arrays.

of DAT over other techniques. On the SS5, we notice an average speedup of 24.2 percent over TSS, 38.0 percent over ESS, and 12.0 percent over LRW. On the SS10, we observe an average speedup of 7.2 percent over TSS, 13.8 percent over ESS, and 4.8 percent over LRW. The speedup on SS10 is smaller because the SS10 has a larger cache, which reduces the number of cache conflicts.

6 Discussion

The main advantage of DAT, the data alignment technique we presented, is its flexibility—decoupling tile selection from the padding phase allows the tile size to be independently optimized without regard to self-interference conflicts. This allows us to select larger tile sizes to maximize the cache utilization. This also allows us to incorporate a user-supplied tile-shape which might be optimized for a specific application. For instance, we can easily handle one-dimensional tiling (SOR example in Table 1), while it cannot be easily incorporated into TSS and LRW. For a given application and cache size, we choose a fixed tile size, independent of the array size (only the pad size varies with the array size), resulting in stability of the performance of the resulting tiled loop. For instance, in the matrix multiplication (Fig. 6), the miss ratio of DAT was in the range [3.11 percent-4.87 percent] in comparison to [2.77 percent-25.18 percent] for LRW, [2.93 percent-23.18 percent] for TSS, and [3.70 percent-23.15 percent] for ESS. Another advantage of our technique is that it is possible to avoid cross-interferences among several tiled arrays (Laplace example in Table 1), while all the other techniques—LRW, TSS, and ESS are targeted at algorithms in which the reuse of only a single array dominates. While accesses to the

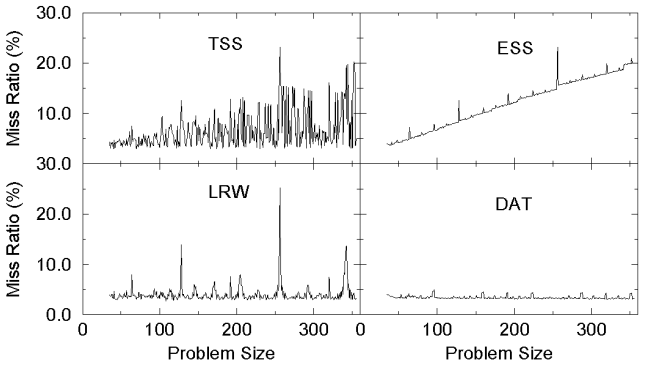


Fig. 6. Matrix multiplication on SS-5: Variation of data cache miss ratio with array dimension.

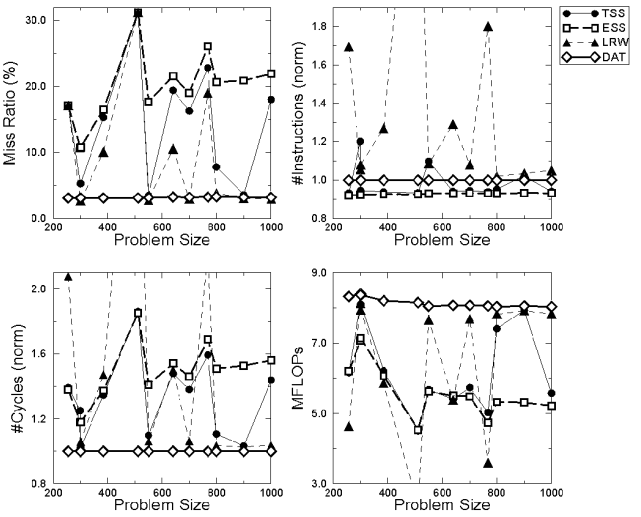


Fig. 7. Performance of L-U decomposition on SS-5: Variation of 1) data cache miss ratio, 2) normalized instruction count, 3) normalized processor cycles, and 4) MFLOPs with array size.

several arrays would cause cross-interference in the other approaches, we are able to completely eliminate cache conflicts arising out of this cross-interference using our technique. Finally, our padding technique can be easily extended to arrays with greater than two dimensions, by serializing the padding procedure over the different dimensions.

One consequence of the padding technique is that the structure of the data arrays is transformed and references to the array in the rest of the code have to reflect the new structure. This, however, does not lead to any performance penalties. For example, when the row size is updated from R to R', the expression to compute the location for $a[i][j]$ changes from $(A + i \times R + j)$ to $(A + i \times R' + j)$, assuming the array begins at A. There could be a conflict in padding sizes if the same array is accessed in different tiled loops. In this case, we invoke the padding strategy on the more critical loop (larger number of array accesses) and, in the others, use one of the existing methods (LRW or TSS) in the remaining loops, with updated row size for the array.

Another consequence of our proposed data alignment technique is that it cannot be directly incorporated into a library routine since the array sizes are not known. The optimization is most effective when it is integrated into the program instead of compiled separately as a library routine. However, the incorporation into a library routine is possible under certain reasonable constraints—for example, if the interface to the tiled array is through file I/O and the library routine constructs the data structure for the arrays in memory.

One way to work around the problem of incorporating the data alignment technique into a library routine is to copy the arrays into a separate memory space with the padding inserted, similar to the *Copy Optimization* technique [6], [13]. However, the overhead of copying the arrays is usually quite significant [13], [8]. In our experiment on the FFT, program copying resulted in a 6 percent improvement in performance over the original code in Fig. 1a, in contrast to the 15 percent improvement for Fig. 1c. It is interesting to

TABLE 1
Performance Results on SS5 and SS10. Data for Problem Sizes 384, 512, 640, and 768 Are Not Included Because That Would Penalize LRW, ESS, and TSS; the Cache Performance Behavior for All Examples Is Similar to That Observed in Fig. 6.

SUN SPARC 5												
Example	MR(x)(%) − MR(DAT)(%)			Inst(x)/Inst(DAT)			Cycle(x)/Cycle(DAT)			Speedup (DAT/x)		
	TSS	ESS	LRW	TSS	ESS	LRW	TSS	ESS	LRW	TSS	ESS	LRW
MM	10.21	22.03	3.76	0.99	0.95	1.07	1.29	1.61	1.19	1.30	1.64	1.14
LUD	6.45	14.18	1.59	1.00	0.93	1.14	1.22	1.40	1.17	1.22	1.40	1.13
SOR	8.81	10.28	0.79	1.03	1.00	1.10	1.31	1.34	1.10	1.23	1.26	1.07
Laplace	6.54	6.22	0.35	1.00	1.00	1.07	1.17	1.16	1.06	1.22	1.22	1.14
SUN SPARC 10												
Example	MR(x)(%) − MR(DAT)(%)			Inst(x)/Inst(DAT)			Cycle(x)/Cycle(DAT)			Speedup (DAT/x)		
	TSS	ESS	LRW	TSS	ESS	LRW	TSS	ESS	LRW	TSS	ESS	LRW
MM	4.01	9.35	1.39	1.00	0.94	1.03	1.15	1.29	1.08	1.15	1.30	1.06
LUD	2.47	4.54	1.06	0.99	0.96	1.08	1.08	1.13	1.12	1.08	1.13	1.09
SOR	0.72	2.09	1.20	1.05	1.00	1.05	1.07	1.09	1.09	1.02	1.05	1.00
Laplace	1.27	1.88	0.03	1.02	1.00	1.04	1.05	1.05	1.03	1.04	1.07	1.04

note the performance improvement of 6 percent in spite of the copy overhead.

7 Conclusions

We presented a data alignment technique DAT for improving the cache performance of numerical applications. We presented an algorithm for selection of tile sizes and data alignment through padding to maximize the utilization of data caches and minimize cache conflicts. We also presented an extension of this approach to reduce cross-interference in applications with multiple tiled arrays.

Our experiments were performed on machines with different cache configurations (SS-5 and SS-10), using a number of standard numerical benchmarks for a range of array sizes. The results demonstrate that our technique results in consistently low data cache miss ratios and effective cache utilization, leading to good overall performance.

Acknowledgments

This work was partially supported by grants from the U.S. National Science Foundation (CDA-9422095) and the U.S. Office of Naval Research (N00014-93-1-1348).

References

[1] A. Milenkovic and V. Milutinovic, "Lazy Prefetching," *Proc. 31st Hawaii Int'l Conf. System Science,* vol. 7, Mauna Lani, Hawaii, Jan. 1998.
[2] M.J. Wolfe, "More Iteration Space Tiling," *Proc. Supercomputing,* pp. 655-664, Reno, Nev., Nov. 1989.
[3] A.R. Lebeck and D.A. Wood, "Cache Profiling and the Spec Benchmarks: A Case Study," *Computer,* vol. 27, no. 10 Oct. 1994.
[4] S. Carr and K. Kennedy, "Compiler Blockability of Numerical Algorithms;" *Proc. Supercomputing,* pp. 114-124, Minneapolis, Minn., Nov. 1992.
[5] M.E. Wolf and M. Lam, "A Data Locality Optimizing Algorithm," *Proc. SIGPLAN '91 Conf. Programming Language Design and Implementation,* pp. 30-44, Toronto, Canada, June 1991.
[6] M. Lam, E. Rothberg, and M.E. Wolf, "The Cache Performance and Optimizations of Blocked Algorithms," *Proc. Fourth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 63-74, Apr. 1991.
[7] K. Esseghir, "Improving Data Locality for Caches," MS thesis, Dept. of Computer Science, Rice Univ., 1993.
[8] S. Coleman and K.S. McKinley, "Tile Size Selection Using Cache Organization and Data Layout," *Proc. ACM SIGPLAN '95 Conf. Programming Language Design and Implementation,* pp. 279-289, La Jolla, Calif., June 1995.
[9] P.M. Embree and B. Kimble, *C Language Algorithms for Digital Signal Processing.* Englewood Cliffs, N.J.: Prentice Hall, 1991.
[10] Sun Microsystems Laboratories, Inc., *Shade User's Manual.* Mountain View, Calif., 1993.
[11] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge Univ. Press, 1992.
[12] P.R. Panda, H. Nakamura, N.D. Dutt, and A. Nicolau, "Improving Cache Performance Through Tiling and Data Alignment," *Solving Irregularly Structured Problems in Parallel,* G. Bilardi, A. Ferreira, R. LÜling, and J. Rolim, eds., vol. 1253. Heidelberg, Germany: Springer-Verlag, 1997.
[13] O. Temam, E.D. 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," *Proc. Supercomputing,* Nov. 1993.

Preeti Ranjan Panda (M'92) received the BTech degree in computer science and engineering from the Indian Institute of Technology, Madras, in 1990, and the MS and PhD degrees in information and computer science from the University of California, Irvine, in 1995 and 1997, respectively. From 1990 to 1993, he worked as a software design engineer at the Design Automation Division of Texas Instruments, India. He is currently with the Advanced Technology Group at Synopsys, Inc., Mountain View, California. His research interests include memory issues in high-level and system-level synthesis, compilation for embedded processors, cache-oriented compiler optimization, and hardware-software codesign. He is the author of the book *Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration* (Kluwer Academic, 1999). He was a recipient of the U.C. Regents Dissertation Fellowship at U.C. Irvine, and an honourable mention award at the Fifth Intl. Conf. on VLSI Design, 1992. He is a member of the IEEE.

Hiroshi Nakamura (M '90) received the BE, ME, and PhD degrees in electrical engineering from the University of Tokyo in 1985, 1987, and 1990, respectively. From 1990 to 1996, he was a faculty member at Institute of Information Sciences and Engineering at the University of Tsukuba, where he was a member of the CP-PACS project. He was a visiting associate professor at the University of California, Irvine, from 1996 to 1997. He is currently an associate professor at the Research Center for Advanced Science and Technology at the University of Tokyo. His research interests include computer architecture, high-performance computing, system-level design assistance, and memory issues in application-specific processors. He received the best paper award from the Information Processing Society of Japan (IPSJ) in 1994. He has served on the program committees of APCHDL94 and PACT97. He is a member of the IEEE, the ACM, the IEICE, and the IPSJ.

Nikil D. Dutt (S'83-M'89-SM'96) received the PhD degree in computer science from the University of Illinois at Urbana-Champaign in 1989. He is currently a professor of ICS and ECE at the University of California at Irvine. His research interests include high-level and system synthesis, system specification languages, retargetable compilers, and hardware/software issues in embedded system design. He is a coauthor of two books: *High-Level Synthesis: Introduction to Chip and System Design* (Kluwer Academic, 1992) and *Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration* (Kluwer Academic, 1999). He received the Best Paper Award at two consecutive Conferences on Hardware Description Languages (CHDL89 and CHDL91). He has served on many conference program committees, including ICCAD, ED&TC, EURODAC, ISSS, and CHDL. He is a member of the IEEE, the ACM, and the IFIP WG 10.5.

Alexandru Nicolau (M'88) received the PhD degree from Yale University, New Haven, Connecticut, in 1984, where he was a member of the ELI/Bulldog project. He is currently a professor of computer science with the University of California, Irvine, where he leads the ESP/PS parallelization project. His research interests are in the areas of fine-grain parallelizing compilers and environments, program transformations, and parallel architectures. Dr. Nicolau is the co-organizer of the Workshop on Languages and Compilers for Parallel Computing and has served on the program committees of the International Conference on Supercomputing and the Symposium on Microarchitecture. He is on the editorial board of the Pitman/MIT Press series of Research Monographs in Parallel and Distributed Computing and is a co-editor-in-chief of the *International Journal of Parallel Programming*. He is a coauthor of the book *Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration* (Kluwer Academic, 1999).