Code Tiling for Improving the Cache Performance of PDE Solvers

Qingguang Huang and Jingling Xue†
School of Computer Science and Engineering
University of New South Wales
Sydney, NSW 2052, Australia

Xavier Vera‡
Institutionen för Datateknik
Mälardalens Högskola
Västerås, Sweden

Abstract

For SOR-like PDE solvers, loop tiling either helps little in improving data locality or hurts their performance. This paper presents a novel compiler technique called *code tiling* for generating fast tiled codes for these solvers on uniprocessors with a memory hierarchy. Code tiling combines loop tiling with a new array layout transformation called *data tiling* in such a way that a significant amount of cache misses that would otherwise be present in tiled codes are eliminated. Compared to nine existing loop tiling algorithms, our technique delivers impressive performance speedups (faster by factors of 1.55 – 2.62) and smooth performance curves across a range of problem sizes on representative machine architectures. The synergy of loop tiling and data tiling allows us to find a problem-size-independent tile size that minimises a cache miss objective function independently of the problem size parameters. This "one-size-fits-all" scheme makes our approach attractive for designing fast SOR solvers without having to generate a multitude of versions specialised for different problem sizes.

1. Introduction

As the disparity between processor and memory speeds continues to increase, the importance of effectively utilising caches is widely recognised. Loop tiling (or blocking) is probably the most well-known loop transformation for improving data locality. This transformation divides the iteration space of a loop nest into uniform tiles (or blocks) and schedules the tiles for execution atomically. Under an appropriate choice of tile sizes, loop tiling often improves the execution times of array-dominated loop nests.

However, loop tiling is known not to be very useful (or even considered not to be needed [15]) for 2D PDE (partial differential equations) solvers. In addition, tile size selection algorithms [5, 6, 12, 14, 18] target only at the 2D arrays accessed in tiled codes. To address these limitations, Song and Li [16] propose a new tiling technique for handling 2D Jacobi solvers. But their technique does not apply to SOR (Successive Over-Relaxation) PDE solvers. Rivera and Tseng [15] apply loop tiling and padding to tile 3D PDE codes. However, they do not exploit a large amount of the temporal reuse carried by the outermost time loop. In this paper, we present a new technique for improving the cache performance of a class of loop nests, which includes multi-dimensional SOR PDE solvers as a special case.

Our compiler technique, called *code tiling*, emphasises the joint restructuring of the control flow of a loop nest through loop tiling and of the data it uses through a new array layout transformation called *data tiling*. While loop tiling is effective in reducing capacity misses, data tiling reorganises the data in memory by taking into account both the cache parameters and the data access patterns in tiled code. By taking control of the mapping of data to memory, we can reduce the number of capacity and conflict misses (which are referred to collectively as *replacement misses*) methodically. In the case of SOR-like PDE solvers assuming a direct-mapped cache, our approach guarantees the absence of replacement misses in every two consecutively executed tiles in the sense that no memory line will be evicted from the cache if it will still be accessed in the two tiles (Theorems 4 and 6). Furthermore, this property carries over to the tiled code we generate for 2D SOR during the computation of all the tiles in a single execution of the innermost tile loop (Theorem 5). Existing tile size algorithms [5, 6, 12, 14, 18] cannot guarantee this property.

The synergy of loop tiling and data tiling allows us to find a problem-size-independent tile size that minimises a cache miss objective function independently of the problem size parameters. This "one-size-fits-all" scheme makes our approach attractive for designing fast SOR solvers for a given cache configuration without having to generate a multitude of versions specialised for different problem sizes.

We have evaluated code tiling for a 2D SOR solver on four representative architectures. In comparison with nine published loop tiling algorithms, our tiled codes have low cache misses, high performance benefits (faster by factors of 1.55 – 2.62), and smooth performance curves across a range of problem sizes. In fact, code tiling has succeeded in eliminating a significant amount of cache misses that would otherwise be present in tiled codes.

The rest of this paper is organised as follows. Section 2 defines our cache model. Section 3 introduces our program model and gives a high-level view of our code tiling strategy. Section 4 describes how to construct a data tiling trans-

†This work is supported by an ARC Grant A10007149.

‡The author was performing part of his PhD studies at UNSW when this work was carried out. He was also supported by the same ARC grant.

formation automatically. Section 5 focuses on finding optimal problem-size-independent tile sizes. Section 6 discusses performance results. Section 7 reviews related work. Section 8 concludes and discusses some future work.

2. Cache Model

In this paper, a data cache is modeled by three parameters: $\mathcal{C}$ denotes its size, $\mathcal{L}$ its line size and $\mathcal{K}$ its associativity. $\mathcal{C}$ and $\mathcal{L}$ are in array elements unless otherwise specified. Sometimes a cache configuration is specified as a triple $(\mathcal{C}, \mathcal{L}, \mathcal{K})$. In addition, we assume a fetch-on-write policy so that reads and writes are not distinguished.

Definition 1 (Memory and Cache Lines) A *memory line* refers to a cache-line-sized block in the memory while a *cache line* refers to the actual block in which a memory line is mapped.

From an architectural standpoint, cache misses fall into one of three categories: *cold*, *capacity*, and *conflict*. In this paper, cold misses are used as before but capacity and conflict misses are combined and called *replacement misses*.

3. Code Tiling

We consider the following program model:

$$
\begin{array}{l}
\textbf{for } I_1 = p_1, q_1 \\
\quad \cdots \\
\textbf{for } I_m = p_m, q_m \\
\quad A(I) = f(A(MI + c_1), \ldots, A(MI + c_\eta))
\end{array}
\tag{1}
$$

where $I = (I_1, \ldots, I_m)$ is known as the *iteration vector*, M is an $n \times m$ integer matrix, the loop bounds p_k and q_k are affine expressions of the outer loop variables $I_1, \ldots, I_{k-1}$, the vectors $c_1, \ldots, c_\eta$ are *offset* integer vectors of length n, and f symbolises some arbitrary computation on the η array references. Thus, A is an n-dimensional array accessed in the loop nest. In this paper, all arrays are in row major. As is customary, the set of all iterations executed in the loop nest is known as the *iteration space* of the loop nest:

$$S = \{I = (I_1, \cdots, I_m) : p_k \leq I_k \leq q_k, k = 1, \ldots, m\} \tag{2}$$

This program model is sufficiently general to include multi-dimensional SOR solvers. Figure 1 depicts a 2D version, where the t loop is called the *time loop* whose loop variable does not appear in the subscript expressions of the references in the loop body. In addition, the linear parts M of all subscript expressions are the identity matrix, and the offset vectors $c_1, \ldots, c_\eta$ contain the entries drawn from $\{-1, 0, 1\}$. These solvers are known as *stencil* codes because they compute values using neighbouring array elements in a fixed stencil pattern. The stencil pattern of data accesses is repeated for each element of the array.

Without loss of generality, we assume that the program given in (1) can be tiled legally by rectangular tiles [19]. For the 2D SOR code, tiling the inner two loops is not beneficial since a large amount of temporal reuse carried by the time loop is not exploited. Due to the existence of the dependence vectors $(1, -1, 0)$ and $(1, 0, -1)$, tiling all three loops by rectangles would be illegal [19]. Instead, we skew the iteration space by using the linear transformation $\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$ and then permute the time step into the innermost position. This gives rise to the program in Figure 2. We choose to move the time step inside because a large amount of temporal reuse in the time step can be exploited for large P.

```
double A(0 : N+1, 0 : N+1)
for t = 0, P-1
for i = 1, N
for j = 1, N
    A(i,j) = 0.2 * (A(i,j) + A(i-1,j) + A(i,j-1)
                 +A(i+1,j) + A(i,j+1))
```

Figure 1. 2D SOR code.

```
double A(0 : N+1, 0 : N+1)
for i = 0, P+N-2
for j = 0, P+N-2
for t = max(0, i-N+1, j-N+1), min(P-1, i, j)
   A(i-t, j-t) = 0.2 * (A(i-t+1, j-t) + A(i-t, j-t+1)
              +A(i-t, j-t)+A(i-t, j-t-1)+A(i-t-1, j-t))
```

Figure 2. Skewed 2D SOR code.

Loop tiling can be understood as a mapping from the iteration space S to $\mathbb{Z}^{2m}$ such that each iteration $(I_1, \ldots, I_m) \in S$ is mapped to a new point in $\mathbb{Z}^{2m}$ [7, 19]:

$$(I_1, \ldots, I_m) \quad \rightarrow \quad (\lfloor \frac{I_1}{T_1} \rfloor, \ldots, \lfloor \frac{I_m}{T_m} \rfloor, I_1, \ldots, I_m) \tag{3}$$

where $(T_1, \ldots, T_m)$ is called the *tile size* and $(\lfloor I_1/T_1 \rfloor, \ldots, \lfloor I_m/T_m \rfloor)$ uniquely identifies the tile that the iteration $(I_1, \ldots, I_m)$ belongs to. Viewed as a loop transformation, loop tiling decomposes an m-dimensional loop into a $2m$-dimensional loop nest, where the outer m loops are the *tile loops* controlling the execution of tiles and the inner m loops are the *element loops* controlling the execution of the iterations in a tile.

Definition 2 (Adjacent Tiles) Two tiles identified by $(u_1, \ldots, u_m)$ and $(u'_1, \ldots, u'_m)$ are said to be *adjacent* if $u_1 = u'_1, \ldots, u_{m-1} = u'_{m-1}$ and $u_m = u'_m - 1$.

Definition 3 (Intra-, Inter$_1$- and Inter$_2$-Tile (Replacement) Misses and Cold Misses) Let u be a given tile and u' be its adjacent tile previously executed. Let there be a total of k accesses, $a_1, \cdots, a_k$, to a memory line ℓ in the tile u. Any of the last $k-1$ such accesses, a_i, where $i > 1$, is a replacement miss if ℓ is found not to be in the cache when a_i is executed. Such a replacement miss is called a *intra-tile*

Algorithm: CodeTiling
INPUT: • A program that conforms to the model (1)
• Cache parameters $(\mathcal{C}, \mathcal{L}, 1)$ (in array elements)
OUTPUT: Tiled code for the given program

(a) Generate the initial tiled code using loop tiling [19].

```
The 2m loops go here // omitted
    A(I) = f(A(MI + c_1), ..., A(MI + c_η))
```

(b) Construct a data tiling transformation $g : \mathbb{Z}^n \to \mathbb{Z}$, to eliminate both intra- and inter$_1$-tile misses (Section 4).
(c) Find the problem-size-independent tile size to miminise the inter$_2$-tile misses as described in Section 5.
(d) Modify the tiled code obtained in (a) to get:

```
The code to copy A to a new 1D array named B
The same loop nest as in (a) goes here // omitted
    B(g(I)) = f(B(g(MI + c_1)), ..., B(g(MI + c_η)))
The code to copy the results in B to A
```

(e) Compute g incrementally using additions/subtractions and apply loop distribution, if necessary, to avoid max and min.

Figure 3. Code tiling for direct-mapped cache.

(replacement) miss. If the access a_1 is a miss, there are three cases. (a) If ℓ was also previously accessed in u', then the miss is called an *inter$_1$-tile (replacement) miss*. (b) If ℓ was previously accessed but not in u', then the miss is called an *inter$_2$-tile (replacement) miss*. (c) Otherwise, the miss is a cold miss as will be classified in the standard manner.

According to this definition, there are four kinds of cache misses in tiled code: cold, intra-tile, inter$_1$-tile and inter$_2$-tile.

Let u and u' be two adjacent tiles. If a tiled loop nest is free of intra- and inter$_1$-tile misses in both tiles, then no memory line will be evicted from the cache during their execution if it will still be accessed in u and u', and conversely.

Figure 3 gives a high-level view of code tiling for direct-mapped caches. In Step (b), we construct a data tiling g to map the n-dimensional array A to the 1D array B such that the tiled code operating on B is free of intra- and inter$_1$-tile misses (Definition 4). There can be many choices for such tile sizes. In Step (c), we choose the one such that the number of inter$_2$-tile misses is minimised. The optimal tile size found is independent of the problem size because our cost function is (Section 5). Finally, our construction of g ensures that the number of cold misses in the tiled code has only a moderate increase (due to data remapping) with respect to that in the original program.

Definition 4 (Data Tiling) Let a direct-mapped cache $(\mathcal{C}, \mathcal{L}, 1)$ be given. A mapping $g : \mathbb{Z}^n \to \mathbb{Z}$ (constructed in Step (b) of Figure 3) is called a *data tiling* if the tiled code given is free of intra- and inter$_1$-tile misses.

For a $\mathcal{K}$-way set-associative cache $(\mathcal{C}, \mathcal{L}, \mathcal{K})$, where $\mathcal{K} > 1$, we treat the cache as if it were the direct-mapped cache $(\frac{\mathcal{K}-1}{\mathcal{K}}\mathcal{C}, \mathcal{L}, 1)$. As far as this hypothetical cache is concerned, g used in the tiled code is a data tiling transformation. By using the effective cache size $\frac{\mathcal{K}-1}{\mathcal{K}}\mathcal{C}$ to model the impact of associativity on cache misses, we are still able to eliminate all intra-tile misses for the physical cache (Theorem 3). In the special case when $\mathcal{K} = 2$, the cache may be under utilised since the effective cache size is only $\mathcal{C}/2$. Instead, we will treat the cache as if it were $(\mathcal{C}, \mathcal{L}, 1)$. The effectiveness of our approach has been validated by extensive experiments conducted (only) on set-associative caches.

4. Data Tiling

In this section, we present an algorithm for automating the construction of data tiling transformations. Throughout this section, we assume a direct-mapped cache, denoted by $(\mathcal{C}, \mathcal{L}, 1)$, where the cache size $\mathcal{C}$ and the line size $\mathcal{L}$ are both in array elements. An application of the results in this section for set-associative caches is discussed in Section 3.

We will focus on a loop nest that conforms to the program model defined in (1) with the iteration space S given in (2). We denote by *offset*(A) the set of the offset vectors of all η array references to A, i.e., *offset*$(A) = \{c_1, \ldots, c_\eta\}$. The notation e_i denotes the i-th elementary vector whose i-th component is 1 and all the rest are 0.

Recall that a loop tiling is a mapping as defined in (3) and that $T = (T_1, \cdots, T_m)$ denotes the tile size used. Let S_T be the set of all the tiles obtained for the program:

$$S_T = \{u = (u_1, \ldots, u_m) : u = (\lfloor \frac{I_1}{T_1} \rfloor, \ldots, \lfloor \frac{I_m}{T_m} \rfloor), I \in S\} \quad (4)$$

Let $T(u)$ be the set of all iterations contained in the tile u:

$$T(u) = \{v = (v_1, \ldots, v_m) : u_k T_k \le v_k \le (u_k + 1)T_k - 1, \; k = 1, \ldots, m\} \quad (5)$$

In this definition, the constraint $I \in S$ from (4) is omitted. Thus, the effect of the iteration space boundaries on $T(u)$ is ignored. As a result, $|T(u)|$ is invariant with respect to u.

For notational convenience, the operator mod is used as both an infix and a prefix operator. We do not distinguish whether a vector is a row or column vector and assume that this is deducible from the context.

Let *addr* be a memory address. In a direct-mapped cache $(\mathcal{C}, \mathcal{L}, 1)$, the address resides in the memory line $\lfloor addr/\mathcal{L} \rfloor$ and is mapped to the cache line $\mathrm{mod}(\lfloor addr/\mathcal{L} \rfloor, \mathcal{C}/\mathcal{L})$.

In Section 4.1, we give a sufficient condition for a mapping to be a data tiling transformation. In Section 4.2, we motivate our approach by constructing a data tiling transformation for the 2D SOR program. Section 4.3 constructs data tiling transformations for the programs defined in (1).

4.1. A Sufficient Condition

For a tile $u \in S_T$, its *working set* (i.e., the set of elements accessed inside u), denoted $D(T(u))$, is given by:

$$D(T(u)) = \{MI + c : I \in T(u), c \in \textit{offset}(A)\} \quad (6)$$

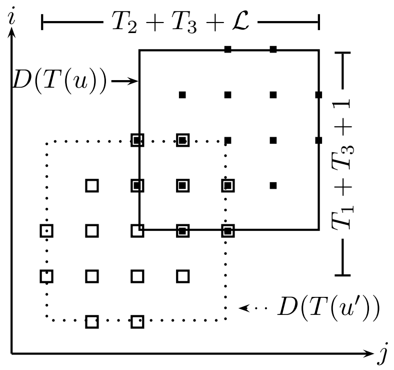


Figure 4. An illustration of Theorem 1 using the 2D SOR in Figure 2. It is assumed $T_1 = T_2 = T_3 = \mathcal{L} = 2$ and u and u' are two arbitrary adjacent tiles executed in that order. Thus, $|T(u)| = |T(u')| = 2\times 2\times 2 = 8$. The corresponding working sets $D(T(u))$ and $D(T(u'))$ are depicted by the solid and dotted (larger) boxes, respectively. Thus, each (small) solid or plain box at (i, j) depicts an array element $A(i, j)$ identified by its array indices (i, j). The two "distance numbers" T_1+T_3+1 and $T_2 + T_3 + \mathcal{L}$ will be referred to in Section 4.2.

It is easy to show that $D(T(u))$ is a translate of $D(T(u'))$ for $u, u' \in S_T$. This property plays an important role in our development, which leads directly to the following result.

Theorem 1 *Let $u, u' \in S_T$ be two adjacent tiles, where $u' = u + e_m$. Then $|D(T(u)) \setminus D(T(u'))| = |D(T(u')) \setminus D(T(u))|$ and $|D(T(u')) \setminus D(T(u))|$ is independent of u.*

Figure 4 illustrates Theorem 1 with the 2D SOR example in Figure 2. This theorem implies that the number of elements that are accessed in u but not in u', i.e., $|D(T(u)) \setminus D(T(u'))|$ is exactly the same as the number of elements that are accessed in u' but not in u, i.e., $|D(T(u')) \setminus D(T(u))|$. If we can find a 1-to-1 mapping $\psi : D(S) \to \mathbb{Z}$ such that $\psi : D(T(u')) \setminus D(T(u)) \to \psi(D(T(u)) \setminus D(T(u'))) \bmod \mathcal{C}$ and use the mapping to map the element $A(MI + c)$ to $B(\psi(MI + c))$, where $c \in$ *offset*(A), then the two corresponding elements in the two sets will be mapped to the same cache line. By convention, $D(S)$ is the union of $D(T(u))$ for all $u \in S_T$. As a result, the newly accessed data in the set $D(T(u')) \setminus D(T(u))$ when u' is executed will evict from the cache exactly those data in the set $D(T(u)) \setminus D(T(u'))$ previously accessed in u.

However, this does not guarantee that all intra- and inter$_1$-tile misses are eliminated. Below we give a condition for data tiling to guarantee these two properties.

For a 1-to-1 mapping $g : \mathbb{Z}^n \to \mathbb{Z}$ and a subset $W \subset \mathbb{Z}^n$, $g : W \to g(W)$ is said to be a *$(\mathcal{C}, \mathcal{L})$-1-to-1 mapping on W* if whenever the following condition

$$\mathrm{mod}(\lfloor \frac{g(w_1)}{\mathcal{L}} \rfloor, \mathcal{C}/\mathcal{L}) = \mathrm{mod}(\lfloor \frac{g(w_2)}{\mathcal{L}} \rfloor, \mathcal{C}/\mathcal{L}) \qquad (7)$$

holds, where $w_1, w_2 \in W$, then the following must hold:

$$\lfloor \frac{g(w_1)}{\mathcal{L}} \rfloor = \lfloor \frac{g(w_2)}{\mathcal{L}} \rfloor \qquad (8)$$

A mapping $g : D(S) \to \mathbb{Z}$ is said to be *$(\mathcal{C}, \mathcal{L})$-1-to-1 on S_T* if g is $(\mathcal{C}, \mathcal{L})$-1-to-1 on $D(T(u))$ for all $u \in S_T$.

Theorem 2 *Let a direct-mapped cache $(\mathcal{C}, \mathcal{L}, 1)$ be given. A mapping $g : D(S) \to \mathbb{Z}$ is a data tiling if g is $(\mathcal{C}, \mathcal{L})$-1-to-1 on S_T.*

Proof. Follows from Definition 4 and the definition of g. ■

Theorem 3 *Consider a $\mathcal{K}$-way set-associative cache $(\mathcal{C}, \mathcal{L}, \mathcal{K})$ with an LRU replacement policy, where $\mathcal{K} > 1$. If a mapping $g : D(S) \to \mathbb{Z}$ is $(\frac{\mathcal{K}-1}{\mathcal{K}}\mathcal{C}, \mathcal{L})$-1-to-1 on S_T, then there are no intra-tile misses in the tiled code from Figure 3.*

Proof. For the g given, there can be at most $\mathcal{K} - 1$ distinct memory lines accessed during the execution of any single tile. By Definition 4, there cannot be any intra-tile misses. ■

In the case of LRU, we tend to reduce also inter$_1$-tile misses by using $\frac{\mathcal{K}-1}{\mathcal{K}}\mathcal{C}$ as the effective cache size.

4.2. Constructing a Data Tiling for 2D SOR

In this section, we construct a data tiling transformation to eliminate all intra- and inter$_1$-tile misses for 2D SOR. We will continue to use the example given in Figure 4. Since the array A is stored in row major, the elements depicted in the same row are stored consecutively in memory. In Step (b) of Figure 3, we will construct a data tiling g to map A to B such that the elements of B will reside in the memory and cache lines as illustrated in Figure 5. (It should be pointed out that g is not a block-cyclic array layout transformation.)

The basic idea is to divide the set of all elements of A into equivalence classes such that $A(i, j)$ and $A(i', j')$ are in the same class if $i = i' \bmod (T_1 + T_3 + 1)$ and $j = j' \bmod (T_2+T_3+\mathcal{L})$. For all array elements of A in the same equivalence class, we will construct g such that their corresponding elements in the 1D array B have the same memory address (modulo $\mathcal{C}$). In other words, $A(i, j)$ and $A(i', j')$ are in the same class iff $g(i, j) = g(i', j') \bmod \mathcal{C}$. In Figure 5, the two elements of A connected by an arc are mapped to the same cache line. This ensures essentially that the unused elements that are accessed in a tile will be replaced in the cache by the newly accessed elements in its adjacent tile to be executed next. As mentioned earlier, this does not guarantee the absence of intra- and inter$_1$-tile misses. To eliminate them, we must impose some restrictions on (T_1, T_2, T_3). For example, a tile size that is larger than the cache size will usually induce intra-tile misses.

In the 2D SOR program given in Figure 2, the linear part of an array reference is defined as follows:

$$M = \begin{pmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \end{pmatrix}$$

and *offset*$(A) = \{c = (c_1, c_2) : |c_1| + |c_2| \le 1\}$.

Let $u = (ii, jj, tt), u' = (ii, jj, tt + 1) \in S_T$ be two adjacent tiles. $T(u)$ and $T(u')$ are defined according to (5).

By Definition 4, it suffices to find a $(\mathcal{C}, \mathcal{L})$-1-to-1 mapping on S_T. To do so, we need a 2-parallelotope containing $D(T(u))$ defined by $\{I \in \mathbb{Z}^2 : -T_3 \le G_1 I \le$

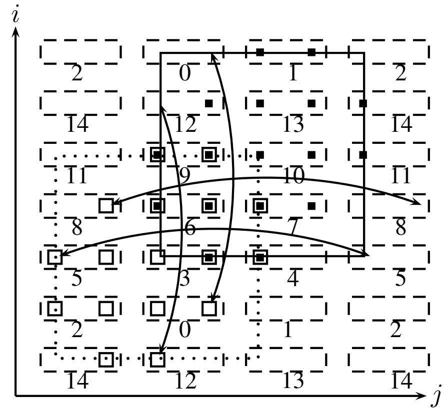


Figure 5. Memory reorganisation effected by a data tiling g when $\mathcal{C}=30$ and $\mathcal{L}=2$. Continuing from Figure 4, g ensures that the elements of A are mapped to B so that all elements in B are aligned at the memory line boundaries as shown. Each dashed box indicates that the two elements inside are in the same memory line; the number below indicates the cache line to which the memory line is mapped. The elements connected by an arc are mapped to the same cache line.

$T_1+T_3, -T_3 \le G_2 I \le T_2+T_3\}$, where $G_1=(0,1)$ and $G_2=(1,0)$. This parallelotope can be obtained by our algorithms FindFacets and FindQ given in Appendix A. We denote by $(G, F(u), K)$ this parallelotope, where $F(u) = (-T_3, -T_3), K = (T_1+T_3, T_2+T_3)$ and the first and second rows of G are G_1 and G_2, respectively. According to $(G, F(u), K)$, we classify the points in the data space $[0, N+1] \times [0, N+1]$ (i.e., the set of array indices of A) and find a mapping such that the points in the same class are mapped into the same cache line. We say that two points (i,j) and (i',j') are *equivalent* if $G_1(i-i', j-j') = s(T_1+T_3+1)$ and $G_2(i-i', j-j') = t(T_2+T_3+\mathcal{L})$, i.e., if $i = i' + s(T_1+T_3+1)$ and $j = j' + t(T_2+T_3+\mathcal{L})$, for some integers s and t. Two points are in the same class iff they are equivalent. (For example, the two points connected by an arc in Figure 5 are in the same equivalence class.)

Let $T_{13} = T_1+T_3+1$ and $T_{23} = \lceil \frac{T_2+T_3+1}{\mathcal{L}} \rceil \mathcal{L}$. We define:

$$g(i,j) = (\lfloor i/T_{13} \rfloor \lceil (N-1)/T_{23} \rceil + \lfloor j/T_{23} \rfloor)\mathcal{C} + \bmod(i, T_{13}) T_{23} + \bmod(j, T_{23}) \tag{9}$$

Theorem 4 *Let a direct-mapped cache $(\mathcal{C}, \mathcal{L}, 1)$ be given. Then g defined in (9) is a data tiling transformation for the 2D SOR if (T_1, T_2, T_3) satisfies the following two conditions:*

1. *$\mathcal{L}$ divides both T_2 and T_3, and*
2. *$(T_1+T_3+1)\lceil \frac{T_2+T_3+1}{\mathcal{L}} \rceil \mathcal{L} \leqslant \mathcal{C}$*

Proof. See Appendix A.

Therefore, g in (9) for 2D SOR guarantees that the tiled code for the program is free of intra- and inter$_1$-tile misses provided the conditions in Theorem 4 are satisfied.

In the example illustrated in Figures 4 and 5, we have $T_1 = T_2 = T_3 = \mathcal{L} = 2$ and $\mathcal{C} = 30$. Both conditions in Theorem 4 are true. The resulting data tiling can be obtained by substituting these values into (9).

In fact, our g has eliminated all inter$_2$-tile misses *among the tiles in a single execution of the innermost tile loop*.

Theorem 5 *Under the same assumptions of Theorem 4, g defined in (9) ensures that during any single execution of the innermost tile loop, every memory line, once evicted from the cache, will not be accessed during the rest of the execution.*

Proof. See Appendix A.

4.3. Constructing a Data Tiling for (1)

We now give a data tiling, denoted g, for a program of the form (1). This time we need an r-parallelotope [17] that contains $D(T(u))$, where r is the dimension of the affine hull of $D(T(u))$. This parallelotope, denoted $\mathcal{P}(T(u))$, is found by our algorithm FindFacets. We can see that $\mathcal{P}(T(u))$ is the smallest r-parallelotope containing $D(T(u))$ if the components of the offset vectors in *offset*(A) are all from $\{-1, 0, 1\}$. Therefore, it is only necessary to map the elements of A that are accessed in the loop nest to B. Hence, g is a mapping from $\mathbb{Z}^r$ to $\mathbb{Z}$, where $r \leqslant n$.

Let $\phi(T)$ and $\psi(T)$ be the number of elements contained in $D(T(u))$ and $\mathcal{P}(T(u))$, respectively. From now on we assume that a tile fits into the cache, i.e., $\psi(T) \le \mathcal{C}$. Let $\mathcal{P}(T(u)) = (G, F(u), K)$ be found by FindFacets and Q by FindQ. Without loss of generality, we assume that $0 \in D(S)$ and $G\tilde{Q}S = \{v = (v_1, \ldots, v_r) : v_i = \lfloor G_i I / Q_i \rfloor, I \in S, i = 1, \ldots, r\}$, where $\tilde{Q} = \mathrm{diag}(\tilde{Q}_1, \ldots, \tilde{Q}_r)$, $\tilde{Q}_1 = Q_1, \cdots, \tilde{Q}_{r-1} = Q_{r-1}$ and $\tilde{Q}_r = \lceil \frac{Q_r}{\mathcal{L}} \rceil \mathcal{L}$. We call $G\tilde{Q}S$ the *data tile space*. For the 2D SOR example, we have $r=2$, $Q = \mathrm{diag}(T_1+T_3+1, T_2+T_3+1)$ and $\tilde{Q} = \mathrm{diag}(T_1+T_3+1, \lceil \frac{T_2+T_3+1}{\mathcal{L}} \rceil \mathcal{L})$. Assume that LB_i and UB_i are the smallest and largest of the i-th components of all the points in $G\tilde{Q}S$, respectively. For $I \in D(S)$, let $y(I) = (y_1(I), \ldots, y_r(I)) = \lfloor \tilde{Q}^{-1} G I \rfloor$ and $z(I) = (z_1(I), \ldots, z_r(I)) = GI - \tilde{Q} y(I)$.

Let $v = (v_1, \ldots, v_r)$ and

$$\mathrm{rowLayout}(v_1, \ldots, v_r) = \Sigma_{j=1}^{r-1} \bmod(G_j v, \tilde{Q}_j) \Pi_{k=j+1}^{r} (K_k+1) + \bmod(G_r v, \tilde{Q}_r) \tag{10}$$

Let

$$g(v_1, \ldots, v_r) = \mathrm{rowLayout}(z_1(v), \ldots, z_r(v)) + \mathcal{C} \Sigma_{j=1}^{r} y_j(v) \Pi_{k=j+1}^{r} (\mathrm{UB}_k - \mathrm{LB}_k) \tag{11}$$

where $\Pi_{k=r+1}^{r}(\mathrm{UB}_k - \mathrm{LB}_k) = 0$.

Theorem 6 *Let a direct-mapped cache $(\mathcal{C}, \mathcal{L}, 1)$ be given. Then g defined in (11) is a data tiling transformation for (1) if the following two conditions are true:*

1. *$\mathcal{L}$ divides $F_r(u)$ for all $u \in S_T$.*
2. *$(\Pi_{k=1}^{r-1} Q_k) \lceil \frac{Q_r}{\mathcal{L}} \rceil \mathcal{L} \leqslant \mathcal{C}$.*

Proof. Under the given two conditions, g is $(\mathcal{C}, \mathcal{L})$-1-to-1 on S_T. By Theorem 2, g is a data tiling as desired.

5. Finding Optimal Tile Sizes

Let a loop nest of the form (1) be given, where A is the array accessed in the nest. Let this loop nest be tiled by the tile size $T = (T_1, \dots, T_m)$. Let $\tilde{T} = (T_1, \dots, T_{m-1}, 2T_m)$. Using the notation introduced in Section 4.3, $\phi(T)$ represents the number of distinct array elements accessed in a tile and $\phi(\tilde{T})$ the number of distinct array elements accessed in two adjacent tiles. Thus, $\phi(\tilde{T}) - \phi(T)$ represents the number of new array elements accessed when we move from one tile to its adjacent tile to be executed next.

Our cost function is given as follows:

$$f(T) \quad = \quad \frac{T_1 \times \cdots \times T_m}{\phi(\tilde{T}) - \phi(T)} \qquad (12)$$

For each tile size that induces no intra- and inter$_1$-tile misses under data tiling, the number of cache misses (consisting of cold and inter$_2$-tile misses) in the tiled code is dominated by $|S_T|/f(T)$. Hence, the optimal tile size is a maximal point of f such that the conditions in Theorem 6 (or those in Theorem 4 for 2D SOR are satisfied). Of all tile sizes without intra- and inter$_1$-tile misses, we therefore take the one such that the number of inter$_2$-tile misses is minimised. Hence, the total number of cache misses is minimised.

The set of all tile sizes is $\{(T_1, \dots, T_m) : 1 \leqslant T_1, \dots, T_m \leqslant \mathcal{C}\}$. The optimal one can be found efficiently by an exhaustive search with a worst-time complexity being $O(\mathcal{C}^m)$, where $\mathcal{C}$ is the cache size in array elements (rather than bytes). (The worst-time complexity when $m = 2$ can be tightened to be $O(\mathcal{C} \log \mathcal{C})$.) Essentially, we simply go through all tile sizes that satisfy the conditions mentioned above and pick the one that is a maximal point of f.

Next we provide a characterisation of cache misses for a program p of the form (1) when $\mathcal{L} = 1$; it can be generalised to the case when $\mathcal{L} > 1$. Let *OMN*(T) be the smallest among the cache miss numbers of all tiled codes for p obtained using the traditional loop tiling under a fixed T but all possible array layouts of A. Let *DTMN*(T) be the cache miss number of the tiled code for p we generate when the layout of A is defined by a data tiling transformation.

Theorem 7 *Let a direct-mapped cache* $(\mathcal{C}, 1, 1)$ *be given. Assume that* $a_1 > 0, \dots, a_m > 0$ *are constants and the iteration space of (1) is* $[0, Na_1 - 1] \times \cdots \times [0, Na_m - 1]$. *Let* T *and* T' *be two tile sizes. If* $\mathcal{P}(T(u)) = D(T(u))$, *then* ***OMN***$(T')$ − ***DTMN***$(T) \geq \Pi_{s=1}^m (Na_s + 1)(\frac{\Pi_{s=1}^m (1 - 2/(Na_s+1))}{f(T')} - \frac{1}{f(T)} - \frac{1}{Na_m+1})$.

Proof. When $\mathcal{L} = 1$, we have the two inequalities:

$$\begin{array}{rcl} \mathit{OMN}(T') & \geq & \Pi_{s=1}^m \lfloor Na_s/T'_s \rfloor (\phi(\tilde{T}) - \phi(T)) \\ & = & \Pi_{s=1}^m \lfloor Na_s/T'_s \rfloor \Pi_{s=1}^n T'_s / f(T') \\ & \geq & \Pi_{s=1}^m (Na_s - 1)/f(T') \\ \mathit{DTMN}(T) & \leq & \Pi_{s=1}^{m-1} (Na_s+1) + \Pi_{s=1}^m (Na_s+1)/f(T) \end{array}$$

which together imply the inequality in the theorem. ■

This theorem implies that when N is large and if we choose T such that $f(T) > f(T')$, then the number of cache misses in our tiled code is smaller than that obtained by loop tiling regardless what array layout is used for the array A.

6. Experimental Results

We evaluate code tiling using the 2D SOR solver and compare its effectiveness with nine loop tiling algorithms on the four platforms as described in Table 1. In all our experiments, the 2D SOR is tiled only for the first level data cache in each platform.

All "algorithms" considered in our experiments are referred to by the following names: `seq` denotes the sequential program, `cot` denotes code tiling, `lrw` is from [18], `tss` from [5], `ess` from [6], `euc` from [14], `pdat` from [12], and `pxyz` is the padded version of `xyz` with pads of 0–8 elements (the same upper bound used as in [14]).

Our tiled code is generated according to Figure 3. The program after its Step (a) is given in Figure 6. The data tiling function g required in Step (b) is constructed according to (9). The problem-size-independent tile sizes on the four platforms are found in Step (c) according to Section 5 and listed in Table 2. Note that in all platforms, the optimal $T_3 = \mathcal{L}$ holds. The final tiled code obtained in Step (d) is optimised as described in Step (e). Note that T_3 does not appear in the tiled code given in Figure 6 since the two corresponding loops are combined by loop coalescing.

Platform	(T_1, T_2, T_3)
Pentium III	$(33, 32, 4)$
Pentium 4	$(15, 16, 8)$
R10K	$(50, 60, 4)$
Alpha 21264	$(76, 80, 8)$

Table 2. Problem-size-independent tile sizes.

All programs are in ANSI C, compiled and executed on the four platforms as described in Table 1. The last two platforms are SGI Origin 2000 and Compaq ES40 with multiple processors. We used only one single processor during our experiments. All our experiments were conducted when we were the only user on these systems.

The SOR kernel has two problem size parameters P and N. In all our experiments except the one discussed in Figure 11, we fix $P = 500$ and choose N from 400 to 1200 at multiples of 57.

Figure 7 shows the performance results on Pentium III. Figure 7(a) plots the individual execution times, showing that all tiled codes run faster than the sequential program except for `ess` at the larger problem sizes. But our tiled codes perform the best at all problem sizes (represented by the curve at the bottom). Figure 7(b) highlights the overall speedups of all tiled codes over the sequential program. This implies that code tiling is faster by factors of 1.98 – 2.62 over the other tiling algorithms, as shown in Figure 7(c).

Figure 8 shows the performance results on Pentium 4. This time, however, loop tiling is not useful as shown in Figure 8(a). Figure 8(b) indicates that neither of the existing tiling algorithms yields a positive performance gain but code tiling attains a speedup of 1.56. Figure 8(c) shows that code tiling is faster than these algorithms by factors of 1.56 – 1.59.

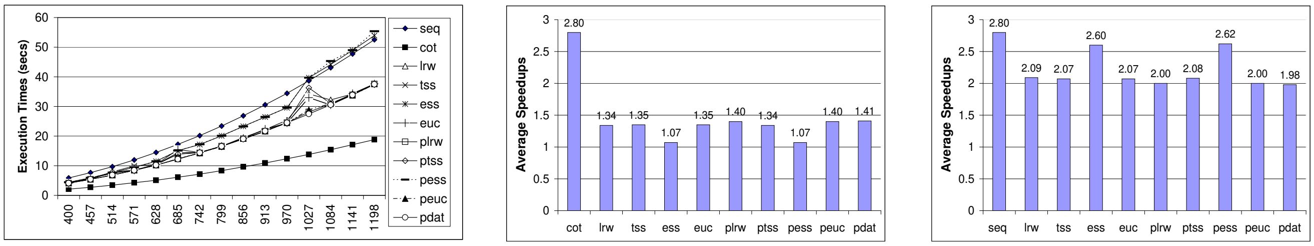
for $ii = 0, (P + N - 2)/T_1$
for $jj = 0, (P + N - 2)/T_2$
for $t = \max(0, ii * T_1 - N + 1, jj * T_2 - N + 1), \min(P - 1, (ii + 1) * T_1 - 1, (jj + 1) * T_2 - 1)$
for $i = \max(ii * T_1, t), \min((ii + 1) * T_1, t + N) - 1$
for $j = \max(jj * T_2, t), \min((jj + 1) * T_2, t + N) - 1$

$$A(i - t, j - t) = 0.2 * (A(i - t + 1, j - t) + A(i - t, j - t + 1) + A(i - t, j - t) + A(i - t, j - t - 1) + A(i - t - 1, j - t))$$

Figure 6. Tiled 2D SOR code.

CPU Clock rate	Pentium III (Coppermine) 933MHz	Pentium 4 1.8GHz	MIPS R10K 250MHz	Alpha 21264 500MHz
L1 D-cache	16KB/32B/4	8KB/64B/4	32KB/32B/2	64KB/64B/2
L1 Replacement Policy	LRU	LRU	LRU	FIFO
L2 D-cache	256KB/32B/8	512KB/128B/8	4MB/128B/2	4MB/64B/4
RAM	256MB	512MB	6GB	6GB
cc version	gcc 3.2.1	gcc 3.2.1	MIPSpro 7.30	DEC C 5.6-075
cc switches	-O2	-O2	-O2	-O2
OS	Debian Linux 3.0	Debian Linux 3.0	IRIX64 6.5	OSF1 4.0

Table 1. Machine configurations.

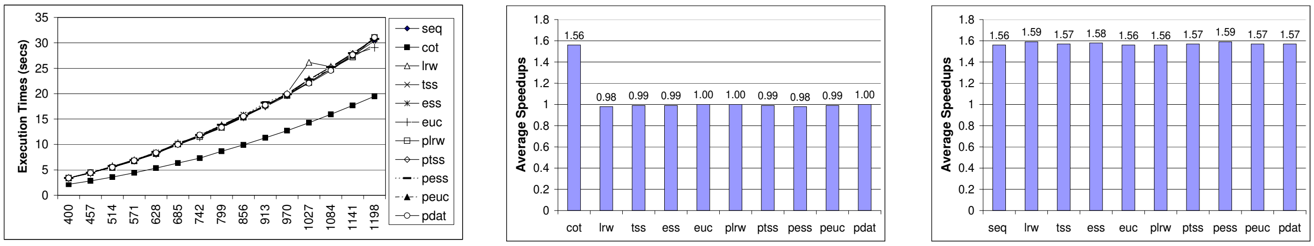


(a) Execution times

(b) Speedups over `seq`

(c) Speedups of `cot` over others

Figure 7. Performance on Pentium III.

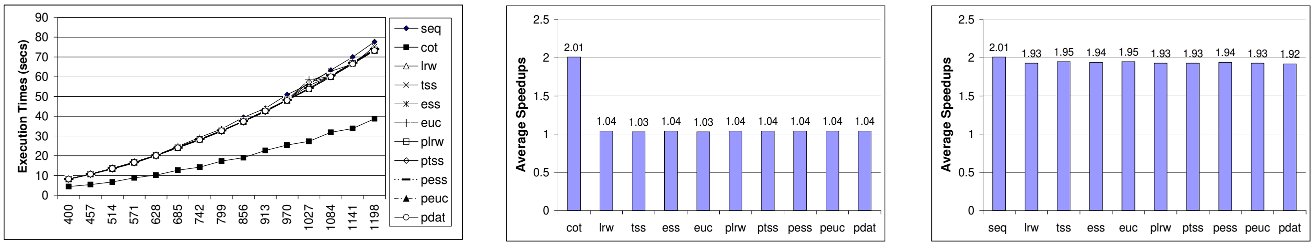


(a) Execution times

(b) Speedups over `seq`

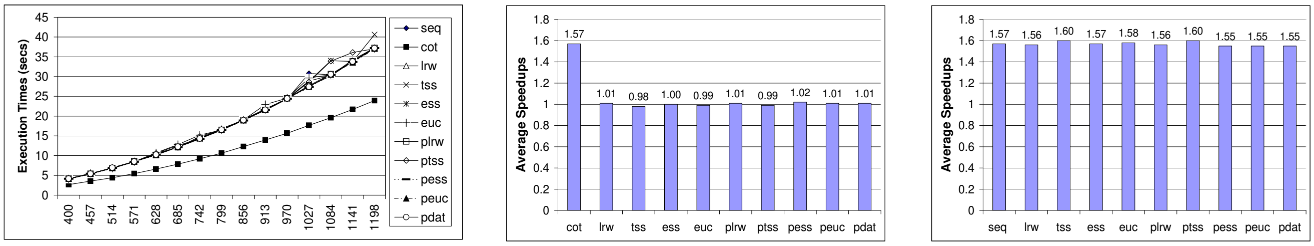
(c) Speedups of `cot` over others

Figure 8. Performance on Pentium 4.



(a) Execution times (b) Speedups over `seq` (c) Speedups of `cot` over others

Figure 9. Performance on R10K.



(a) Execution times (b) Speedups over `seq` (c) Speedups of `cot` over others

Figure 10. Performance on Alpha 21264.

Figure 9 shows the performance results on R10K. Loop tiling helps little. But code tiling achieves a speedup of 2.01, which is in sharp contrast to the negligible positive speedups from the other tiling algorithms. Overall, code tiling is faster by factors of 1.92 – 1.95 over the other algorithms.

Figure 10 shows the performance results on Alpha 21264. Similar trends as in Pentium 4 can be observed. Code tiling is faster than the other algorithms by factors of 1.55 – 1.60.

Some other properties about code tiling are in order.

Copy Cost. All execution times include the copy overheads. In the tiled code for 2D SOR, the copy cost contributes only $O(1/P)$ to the overall time complexity, where P is the number of time steps. We measured the copy cost to be 0.8% – 1.2% on Pentium III, 0.1 – 1.5% on Pentium 4, 0.1 – 1.0% on R10K and 0.1 – 1.3% on Alpha 21264 of the total execution time.

Address Calculation Cost. The data tiling functions used involve integer division and remainder operations and is thus expensive. They are efficiently computed by using incremental additions/subtractions and distributing loop nests to avoid excessive max/min operations.

High and Smooth Performance Curves. Figures 7(a) – 10(a) show clearly that code tiling enjoys high, smooth performance curves across a range of problem sizes on four platforms. To see our stability advantage further, Figure 11 plots the time differences $T(N)-T(N-3)$ between two adjacent problem sizes at multiples of 3.

Space Requirement. The size of the 1D array B introduced in in Figure 3 is given by $g(N,N)$ in (9). For the 2D SOR, we find that $g(N,N) \leqslant N^2 + N\sqrt{\mathcal{C}} + \mathcal{C}$,

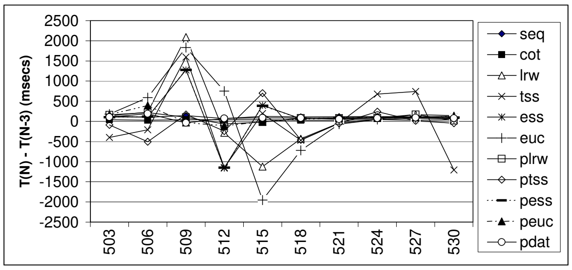


Figure 11. Performance stability on Pentium III ($P = 500$).

where $\mathcal{C}$ is the cache size in terms of array elements rather than bytes. For the problem sizes used in our experiments, $g(N,N)$ ranges from $1.03N^2$ to $1.16N^2$. Note that the multiplier is only 1.33 when $N = 100$. When N is even smaller, tiling is usually not needed. The tiling technique for the Jacobi solvers [16] employs array duplication to remove anti and output dependences. So their constant multiplier is 1.

Cache Misses. To support our claim that code tiling has eliminated a large amount of cache misses present in the tiled codes generated by loop tiling, we evaluated cache performance for all codes involved in our experiments using `PCL` [13]. Figure 12 plots the real L1 data cache misses for all methods on Pentium III. In comparison with Figure 7(a), the significant performance gains correlate well with the significant cache miss reductions at most problem sizes. Note that `lrw` has comparable or even smaller cache miss numbers at some problem sizes. This is because in our tiled codes, some temporaries are required to enable incremental computation of the data tiling function (see Step (d) in Figure 3) and they are not all kept in registers due to a small number of registers available on the `x86` architecture. Despite of this problem, `cot` outperforms `lrw` at all problem sizes. This can be attributed to several reasons (e.g., TLB and L2 misses).

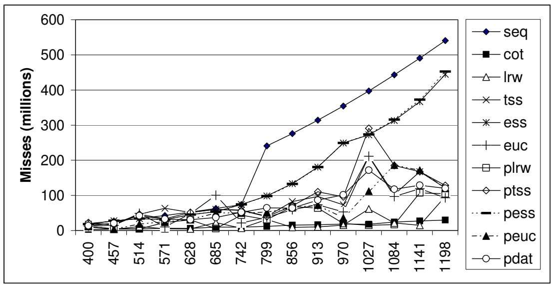


Figure 12. L1 data cache misses on Pentium III.

7. Related Work

To the best of our knowledge, we are not aware of any previous work on applying a *global* data reorganisation strategy to minimise the cache misses in tiled codes. Some earlier attempts on partitioning the cache and mapping arrays into distinct cache partitions can be found in [2, 10]. Manjikian *et al* [10] allocate arrays to equal-sized regions. Chang *et al* [2] allow varying-sized regions but assume all arrays to be one-dimensional. These techniques cannot handle the 2D SOR solvers since these kernels each use one single array — there is nothing to partition.

Compiler researchers have applied loop tiling to enhance data locality. Several tile size selection algorithms [5, 6, 12, 14, 18] find tile sizes to reduce the cache misses in tiled codes. Since these algorithms rely on the default linear layouts of the arrays, padding has been incorporated by many algorithms to help loop tiling stabilise its effectiveness [12, 14].

While promising performance gains in many programs, loop tiling is not very useful for 2D PDE solvers and may even worsen their performance as shown by our experiments. In recognising this limitation, Song and Li [16] present a tiling technique for handling 2D Jacobi solvers. This paper contributes a new technique for improving the performance of multi-dimensional SOR solvers. Rivera and Tseng [15] extend their previous work [14] to 3D solvers but they do not exploit a large amount of temporal reuse carried at the time step as we do here.

Kodukula *el al* [9] propose a data shackling technique to tile imperfect loop nests. But this technique itself does not tell which tile size to use. Like loop tiling, data shackling is a loop transformation. As such, it does not modify the actual layouts of the arrays used in tiled codes.

Chatterjee *et al* [3] consider nonlinear array layouts and achieve impressive performance speedups in some benchmarks when they are combined with loop tiling. However, their technique is orthogonal to loop tiling; they rely on a tile size selection algorithm to find appropriate tile sizes. In addition, they choose nonlinear layouts for all the arrays without making any attempt in partitioning and reorganising them in memory. In other words, they do not directly aim at reducing the cache misses in tiled codes. This may partially explain why they obtain increased cache misses in some benchmarks (due to conflict misses between tiles).

The importance of combining data transformations with loop transformations was recognised earlier [4]. Subsequently, several researchers [8, 11] permit the co-existence of different array layouts (row major, column major, diagonal or others) in a kernel or program-wise and obtain moderate performance gains for benchmark programs.

The PhiPAC project [1] uses an exhaustive search to produce highly tuned tiled codes for specific level-3 BLAS kernels, which are specialised not only for a given cache configuration but also a given problem size. Our code tiling methodology generates automatically a single "optimised" version for an SOR PDE solver for all problem sizes.

8. Conclusion

We have presented a new compiler technique for improving the performance of a class of programs that includes multi-dimensional SOR PDE solvers as a special case. Code tiling combines loop tiling with data tiling in order to reduce cache misses in a predictable and methodical manner. We have evaluated its effectiveness using the classic 2D SOR solver – for which loop tiling is ineffective – on four representative architectures. Our experimental results show that code tiling has eliminated a significant amount of cache misses that would otherwise be present in tiled codes. This translates to impressive performance speedups over nine loop tiling algorithms for a range of problem sizes.

We believe that code tiling can be generalised to other programs, at least to dense matrix codes for which loop tiling is an appropriate means of control flow restructuring for data locality. This will have the potential to eliminate a significant amount of conflict misses still present in tiled codes. Some preliminary results we have obtained on matrix multiplication are extremely encouraging.

References

[1] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology. In *International Conference on Supercomputing*, pages 340–347, 1997.

[2] C.-Y. Chang, J.-P. Sheu, and H.-C. Chen. Reducing cache conflicts by multi-level cache partitioning and array elements mapping. In *7th International Conference on Parallel and Distributed Systems (ICPADS'00)*, Iwate, Japan, 2000.

[3] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear array layout for hierarchical memory systems. In *ACM International Conference on Supercomputing (ICS'99)*, pages 444–453, Rhodes, Greece, Jun. 1999.

[4] M. Cierniak and W. Li. Unifying data and control transformations for distributed shared memory machines. In *ACM*

SIGPLAN'95 Conference on Programming Language Design and implementation, California, 1995.

[5] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI'95)*, pages 279–290, Jun. 1995.

[6] K. Esseghir. Improving data locality for caches. M.S. thesis, Rice University, Dept. Computer Science, 1993.

[7] F. Irigoin and R. Triolet. Supernode partitioning. In *15th Annual ACM Symposium on Principles of Programming Languages*, pages 319–329, San Diego, California., Jan. 1988.

[8] M. Kandemir and J. Ramanujam. A layout-concious iteration space transformation technique. *IEEE Transactions on Computers*, 50(12):1321–1335, Dec. 2001.

[9] I. Kodukul, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *ACM SIGPLAN '97 Conference on Programming Language Design*, pages 346–357, 1996.

[10] N. Manjikian and T. Abdelrahman. Array data layout for the reduction of cache conflicts. In *8th Int. Conf. on Parallel and Distributed Computing Systems*, 1995.

[11] M. F. P. O'Boyle and P. M. W. Knijnenburg. Integrating loop and data transformations for global optimisation. In *International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, 1998.

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

[13] PCL. The Performance Counter Library Version 2.2, 2003. http://www.fz-juelich.de/zam/PCL.

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

[15] G. Rivera and C.-W. Tseng. Tiling optimizations for 3D scientific computations. In *Supercomputing '00*, 2000.

[16] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *ACM SIGPLAN'99 Conference on Programming Language Design and Implementation (PLDI'99)*, pages 215–228, May 1999.

[17] R. Webster. *Convexity*. Oxford University Press, 1994.

[18] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *ACM SIGPLAN'91 Conf. on Programming Language Design and Implementation*, Jun. 1991.

[19] J. Xue. On tiling as a loop transformation. *Parallel Processing Letters*, 7(4):409–424, 1997.

Appendix A

Algorithm FindFacets
Input: M and *offset*(A) in (1), T and $T(u)$
Output: The facets $(G, F(u), K)$ of $\mathcal{P}(T(u))$

For a subset σ of $[1, m]$ and $x = (x_1, \ldots, x_m)$, we denote $x_\sigma = \Sigma_{i \in \sigma} x_i e_i$ and $x^\sigma = \Sigma_{i \in [1,m] \setminus \sigma} x_i e_i$.

1. Calculate the rank of M. Let $r = \text{rank}(M)$. Let $M = [M_1, \ldots, M_m]$, where M_i is the i-th column of M.
2. $\tilde{K} = \{\{k_1, \ldots, k_s\} : 1 \leq k_1 < \cdots < k_s \leq m, \text{rank}([M_{k_1}, \ldots, M_{k_s}]) = r - 1\}$.
3. $\max \tilde{K} = \{\sigma \in \tilde{K} : \nexists\, \sigma' \in \tilde{K}$ such that $\sigma \subset \sigma', \sigma \neq \sigma'\}$.
4. For $\sigma = \{\sigma_{i_1}, \ldots, \sigma_{i_s}\} \in \max \tilde{K}$, find a $G_\sigma \in \mathbb{R}^m$ such that $G_\sigma M_{\sigma_i} = 0, i = 1, \ldots, s, G_\sigma \neq 0$ and G_σ is a linear combination of $M_1, \ldots, M_m$.
5. Let $t_i = u_i T_i$ and $t'_i = (u_i + 1)T_i - 1$. Let *node*$(T(u)) = \{x = (x_1, \ldots, x_m) : x_i \in \{t_i, t'_i\}\}$. For $\sigma \in \max \tilde{K}$, find $x, z \in$ *node*$(T(u))$ such that $G_\sigma M x^\sigma \leq G_\sigma M y^\sigma \leq G_\sigma M z^\sigma$ for all $y \in$ *node*$(T(u))$. Put $\sigma_- = x^\sigma$ and $\sigma_+ = z^\sigma$. Then $MT(u)^\pm_\sigma$ is a facet of $MT(u)$, where $T(u)^\pm_\sigma = \{y_\sigma + \sigma_\pm : y \in T(u)\}$.
6. Let $\max \tilde{K} = \{\sigma^1, \ldots, \sigma^p\}$, $G_i = G_{\sigma^i}$ and $T_i^\pm = T(u)^\pm_{\sigma_i}$. Let $\Gamma(M) = \{\gamma = (\gamma_1, \ldots, \gamma_r) : 1 \leq \gamma_1 < \cdots < \gamma_r \leq p\}$. For $\gamma \in \Gamma(M)$, we have an r-parallelotope $(G^\gamma, F^\gamma, K^\gamma)$ containing $MT(u)$, where $(G^\gamma, F^\gamma, K^\gamma) = \{y \in H : F_i^\gamma \leq G_{\gamma_i} y \leq F_i^\gamma + K_i^\gamma\}$, $F_i^\gamma = G_{\gamma_i} MT^-_{\gamma_i}$ and $K_i^\gamma = G_{\gamma_i} MT^+_{\gamma_i} - G_{\gamma_i} MT^-_{\gamma_i}$. Find a γ such that the volume of $(G^\gamma, F^\gamma, K^\gamma)$ is not smaller than the volume of $(G^{\gamma'}, F^{\gamma'}, K^{\gamma'})$ for all $\gamma' \in \Gamma(M)$.
7. Let $(G^\gamma, F^\gamma, K^\gamma)$ be found in Step 6. Find $c_-^i, c_+^i \in$ *offset*(A) such that $G_i^\gamma c_-^i \leq G_i^\gamma c \leq G_i^\gamma c_+^i$ for all $c \in$ *offset*(A).
8. An r-parallelotope $(G, F(u), K)$ containing $\cup_{c \in \textit{offset}(A)} (MT(u) + c)$ is found, where $G = G^\gamma, F(u) = (F_1, \ldots, F_r), K = (K_1, \ldots, K_r)$ and $F_i = F_i^\gamma + G_i^\gamma c_-^i, K_i = K_i^\gamma + G_i^\gamma c_+^i - G_i^\gamma c_-^i$.

Algorithm FindQ
Input: M and *offset*(A) in (1), $T(u)$ and $(G, F(u), K)$ of $\mathcal{P}(T(u))$
Output: Q

1. Put $G_i = G_{\sigma^{\gamma_i}}$, where σ^{γ_i} is defined Step 6 of FindFacets. Let $1 \leq m_0 \leq m$ and $1 \leq r_0 \leq r$ such that $G_i M_{m_0} \neq 0, i = 1, \ldots, r_0$, $G_i M_{m_0} = 0, i = r_0 + 1, \ldots, r$ and $G_i M_s = 0, i = 1, \ldots, r, s = m_0 + 1, \ldots, m$. Let $\{G_i M y^{\sigma^{\gamma_i}} + G_i c : y \in T(u), c \in \textit{offset}(A)\} = \{h_1^i, \ldots, h_q^i\}$, where $h_1^i < \cdots < h_q^i$.
2. Let $\Delta = t'_{m_0} - t_{m_0} + 1$. If $G_i M_{m_0} > 0$, then take the smallest h_j^i such that $h_j^i + \Delta G_i M_{m_0} > h_q^i$ and define $Q_i = h_j^i + \Delta G_i M_{m_0} - h_1^i$. Otherwise, take the largest h_j^i such that $h_j^i + \Delta G_i M_{m_0} < h_1^i$ and define $Q_i = h_q^i - (h_j^i + \Delta G_i M_{m_0})$.

Proof of Theorem 4:
By Theorem 2, we only need to prove that g is $(\mathcal{C}, \mathcal{L})$-1-to-1 on S_T. Let $u = (ii, jj, tt)$ and $T(u) = \{(i, j, t) : iiT_1 \leq i < (ii+1)T_1, jjT_2 \leq j < (jj+1)T_2, ttT_3 \leq t < (tt+1)T_3\}$. Thus, $D(T(u)) = \{(i - t, j - t) + c : (i, j, t) \in T(u), c \in \textit{offset}(A)\}$. Let $(G, F(u), K)$ be the 2-parallelotope containing $D(T(u))$. Then $F(u) = (iiT_1 - ttT_3, jjT_2 - ttT_3)$. Suppose that $\text{mod}(\lfloor g(i,j)/\mathcal{L} \rfloor, \mathcal{C}/\mathcal{L}) = \text{mod}(\lfloor g(i',j')/\mathcal{L} \rfloor, \mathcal{C}/\mathcal{L})$, where $(i, j), (i', j') \in D(T(u))$. By the second hypothesis, we have that $\lfloor \text{mod}(j, T_{23})/\mathcal{L} \rfloor, \lfloor \text{mod}(j', T_{23})/\mathcal{L} \rfloor < \mathcal{C}$. Thus, $\text{mod}(i, T_{13})(T_{23}/\mathcal{L}) + \lfloor \text{mod}(j, T_{23})/\mathcal{L} \rfloor = \text{mod}(i', T_{13})(T_{23}/\mathcal{L}) + \lfloor \text{mod}(j', T_{23})/\mathcal{L} \rfloor$. Since $\lfloor \text{mod}(j, T_{23})/\mathcal{L} \rfloor < T_{23}/\mathcal{L}$ and $\lfloor \text{mod}(j', T_{23})/\mathcal{L} \rfloor < T_{23}/\mathcal{L}$, we have that $i = i'$, and hence, that $\lfloor \text{mod}(j, T_{23})/\mathcal{L} \rfloor = \lfloor \text{mod}(j', T_{23})/\mathcal{L} \rfloor$. By the first hypothesis that $\mathcal{L}$ divides both T_2 and T_3, $\mathcal{L}$ must divide $g(F(u))$ for all $u \in S_T$. Clearly, $\mathcal{L}$ divides $g(F(u) + (0, T_{23}))$ since $\mathcal{L}$ divides T_{23}. Since $g(F(u)) \leq g(i, j), g(i, j') < g(F(u) + (0, T_{23}))$ and $g(F(u) + (0, T_{23})) - g(F(u)) < \mathcal{C}$, we have $\lfloor g(i,j)/\mathcal{L} \rfloor = \lfloor g(i',j')/\mathcal{L} \rfloor$. Hence, we have proved that g is $(\mathcal{C}, \mathcal{L})$-1-to-1 on S_T. ■

Proof of Theorem 5:
Let $u = (ii, jj, tt)$ be an arbitrary tile and $D(T(u))$ be defined as in the proof of Theorem 3. Let $u' = (ii, jj, tt + 1)$ and $u'' = (ii, jj, tt + m)$, where $m \geq 1$, and $\tilde{D}(T(u'))$ and $\tilde{D}(T(u''))$ be similarly defined. Let $\tilde{D}(T(u))$ be the set of memory lines ℓ such that $\ell \in \tilde{D}(T(u))$ iff there is $(i, j) \in D(T(u))$ such that $B(g(i, j))$ resides in ℓ. From the proof of Theorem 4 we see that g is $(\mathcal{C}, \mathcal{L})$-1-to-1. Thus, any memory line that is accessed in a tile cannot be evicted from the cache in that tile. If a memory line $\ell \in \tilde{D}(T(u))$ is evicted from the cache when the tiles u and u' are executed, we prove next that ℓ must be contained in $\tilde{D}(T(u)) \setminus \tilde{D}(T(u'))$. In fact, if $\ell \notin \tilde{D}(T(u)) \setminus \tilde{D}(T(u'))$, then $\ell \in \tilde{D}(T(u'))$. Suppose that $\ell \in \tilde{D}(T(u'))$. Then $\ell \in \tilde{D}(T(u)) \cap \tilde{D}(T(u'))$, and hence, ℓ cannot be evicted from the cache in either u or u'. This contradicts to the assumption on ℓ. Since $T_3 = \mathcal{L}$, $\mathcal{L}$ divides T_2 and T_{23}, and by noting (9), it is easy to see that $(\tilde{D}(T(u)) \setminus \tilde{D}(T(u'))) \cap \tilde{D}(T(u'')) = \emptyset$. Combining this with that $\ell \in \tilde{D}(T(u)) \setminus \tilde{D}(T(u'))$ shows that ℓ will not be accessed in u''. ■