Automatic Tiling of Iterative Stencil Loops

Zhiyuan Li and Yonghong Song
Purdue University

Iterative stencil loops are used in scientific programs to implement relaxation methods for numerical simulation and signal processing. Such loops iteratively modify the same array elements over different time steps, which presents opportunities for the compiler to improve the temporal data locality through loop tiling. This paper presents a compiler framework for automatic tiling of iterative stencil loops, with the objective of improving the cache performance. The paper first presents a technique which allows loop tiling to satisfy data dependences in spite of the difficulty created by imperfectly-nested inner loops. It does so by skewing the inner loops over the time steps and by applying a uniform skew factor to all loops at the same nesting level. Based on a memory cost analysis, the paper shows that the skew factor must be minimized at every loop level in order to minimize cache misses. A graph-theoretical algorithm, which takes polynomial time, is presented to determine the minimum skew factor. Furthermore, the memory-cost analysis derives the tile size which minimizes capacity misses. Given the tile size, an efficient and general *array-padding* scheme is applied to remove conflict misses. Experiments are conducted on sixteen test programs and preliminary results show an average speedup of 1.58 and a maximum speedup of 5.06 across those test programs.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*compilers*; *optimization*

General Terms: Languages, Performance

Additional Key Words and Phrases: Caches, loop transformations, optimizing compilers

1. INTRODUCTION

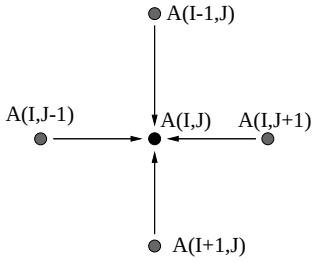
Due to the widening gap between the processor speed and the memory bandwidth, the importance of efficient use of caches is widely recognized [Burger et al. 1996; Ding and Kennedy 2001]. *Loop tiling*, which is also known as *loop blocking*, is a well-known loop transformation to improve the temporal locality, and hence the cache utilization, of a loop nest [Wolfe 1995]. Extensive work has been published previously on how to tile a class of loops often known as *linear algebra* loops, which include familiar examples such as matrix multiplication and Gausian elimination. Unfortunately, the methods developed previously cannot be effectively applied to another important class of loops, which we call *iterative stencil loops*, due to their different data-dependence characteristics.

In this paper, we investigate how to automatically tile iterative stencil loops. This

Author's address: Department of Computer Sciences, Purdue University, West Lafayette, IN 47907. email: {li,songyh}@cs.purdue.edu.

```
      DO T = 1, ITMAX
L1:     DO J1 = 2, N - 1
          DO I1 = 2, N - 1
            temp(I1, J1) = (A(I1 + 1, J1)
              + A(I1 - 1, J1) + A(I1, J1 + 1)
              + A(I1, J1 - 1))/4
          END DO
        END DO
L2:     DO J2 = 2, N - 1
          DO I2 = 2, N - 1
            A(I2, J2) = temp(I2, J2)
          END DO
        END DO
      END DO
```

(a) Jocobi kernel code



(b) Stencil illustration of Jacobi code

Fig. 1. The example of Jacobi

class of loops is used to implement relaxation methods in scientific applications such as numerical simulations and signal processing. In relaxation methods, each matrix element is updated based on the values of its neighboring elements, with the updates repeated over a number of time steps. Thus, there exists a high potential to reuse cached data.

If, however, the total size of the data exceeds the cache size, then executing the stencil loops in the original iteration order will result in repeated cache misses in every time step. Take the Jacobi program kernel (Figure 1(a)) as an example.[1] The outermost loop iterates over the time steps. In every time step T, each $A(I, J)$ element is updated based on the old values of its four neighbors. The computation stencil is shown in Figure 1(b). The *spatial loops*, indexed by J_1, I_1, J_2 and I_2, sweep through nearly $2 \times (N-1)^2$ distinct array elements in each time step. With sufficiently large N, the cache will overflow, causing cache misses in every T iteration when the arrays A and *temp* are referenced.

To apply the idea of loop tiling, the iteration space of the spatial loops is partitioned into *loop tiles*, or simply *tiles*, such that a tile is executed repeatedly over different time steps before the next tile is executed. The tile size can be chosen so that its array footprint, also known as the *array tile*, fits in the cache. If a given iterative stencil loop is perfectly nested, then previous work has shown how to tile it [Wolf 1992]. One can first use *skewing* to transform it into a *fully permutable* loop nest, which can then be safely tiled without additional transformations.

A perfect loop nest has a unique innermost loop which contains all the assignment statements in the loop nest. Unfortunately, iterative stencil loops are usually imperfectly nested. For example, the Jacobi program kernel shown in Figure 1(a) has two separate inner loop subnests, labeled L_1 and L_2, within the *time-step* loop. Due to the data dependences between L_1 and L_2, one cannot skew those two subnests independently. How to deal with these difficulties remained elusive until we presented a method to tile imperfectly-nested iterative stencil loops [Song and Li 1999].

Since then, several authors have published new contributions to the solution

[1]The convergence test is temporarily removed from this example for simplicity of the initial discussion. All code examples in this paper are written in a Fortran-like style, assuming column-major array allocation.

of similar problems. A scheme called *time skewing* performs a value-based flow analysis to optimize for memory locality [Wonnacott 2002]. This scheme, like our previous work, applies to one-dimensional tiles only. Methods such as *loop embedding* [Ahmed et al. 2000] and *iteration space slicing* [Pugh and Rosser 1999] use various heuristics to find legal fusions which, when combined with loop skewing, can improve data reuse in iterative stencil loops. All methods proposed so far require certain re-alignment between different spatial loop subnests. However, the interaction between the loop alignment and loop skewing has not been studied. Hence, how to optimally align spatial loops in order to minimize cache misses is poorly understood. Furthermore, the impact of loop skewing on the optimal selection of tile sizes has not been studied.

In this paper, we develop a theoretical foundation to address those unsolved issues in a comprehensive way. We also present a more extensive set of experimental results to evaluate the effectiveness of applying skewed tiling to iterative stencil loops. The theoretical foundation will be developed in several steps. We first define the general construct of iterative stencil loops before and after tiling. We allow multi-dimensional tiles, and we determine the exact shape of the loop construct after tiling based on parameters such as the *skew factors* and the tile sizes at different loop levels. We develop a number of formulas to estimate the memory reference cost as a function of these parameters. These formulas will show that, in order to minimize the memory reference cost, the *skew factors* must be minimized at all loop levels. We then present an algorithm to determine the minimum permissible skew factors. The cost formulas also give the optimal tile size, assuming the absence of conflict misses in the cache. We then present an efficient *array padding* scheme to eliminate such conflict misses, given a multidimensional tile.

In this paper, we also discuss a number of techniques to reduce skew factors, including *array duplication*, *compatible-loop recognition*, and *circular loop skewing*. Finally, we devise a method to apply tiling to loops with premature exits. This is done by speculatively executing a number of iterations of the time-step loop, with roll-back statements inserted to insure correct program results in case of misspeculation.

The rest of the paper is organized as follows. In Section 2, we present an overview of the tiling mechanism for iterative stencil loops and define the loop construct before and after tiling. This is followed by a presentation of the memory-cost estimation formulas (Section 3), an algorithm to determine the minimum skew factors (Section 4), and an array padding scheme to eliminate reference interferences (Section 5). In Section 6, we discuss enhancements to the main tiling scheme. We then present experimental results in Section 7. Related work is discussed in Section 8, which is followed by a conclusion in Section 9.

2. OVERVIEW AND PRELIMINARIES

In this section, we present an overview of our skewed-tiling scheme and define several basic concepts. We describe the loop construct before and after tiling.

2.1 Overview

Iterative stencil loops which implement relaxation methods for scientific computing generally take the form shown in Figure 2(a). (Although the loops may contain

multiple arrays, only a representative array A is listed. Expressions e_i may read or write A elements.) The outermost T-loop iterates over the time steps, and the T-loop body contains a number of inner loops indexed by $L_{i,k}$. These are called *spatial loops* and they are arbitrarily nested. The second subscript k indicates the nesting level of $L_{i,k}$ among all the spatial loops, and the first subscript i assigns a sequence number among all spatial loops at level k. The total number of spatial loops at the level k is denoted by $m[k]$.

We consider tiling the spatial loops at the top n levels, where n is to be determined by the memory-reference cost analysis (*c.f.* Section 3) and by any other considerations which the compiler writer may have. Thus, Figure 2(a) omits spatial loops that are nested deeper than n. The lower and upper limits of any loop $L_{i,k}$ can be arbitrary expressions. Without loss of generality, all index steps are assumed to be 1. Any dangling statements between two neighboring loop subnests can be moved into one of the two neighbors. Conditional branches are inserted to insure that the dangling statements are executed at the correct moment. Alternatively, one may artificially create a loop subnest, which has iteration counts of one at all loop levels, to enclose such dangling statements.

For the loop nest described above, tiling can be performed with or without first fusing the spatial loops into a single perfect nest. To legally fuse the spatial loops, certain loop bounds must be *realigned* and array subscripts must be modified in order to satisfy the data dependences between the statements which previously existed in the different loop subnests [Pugh and Rosser 1999; Ahmed et al. 2000; Song et al. 2001; Ding and Kennedy 2001]. Fusing the loops before tiling has several drawbacks. The changes to the array indices tend to make it more difficult for the back-end compiler to perform optimizations such as software pipelining on the tiled loops. Furthermore, the increased innermost loop body may cause problems in register allocation. Alternatively, one can choose to tile the loops directly, without fusion, by transforming the given loop nest into the form shown in Figure 2(b). The loop bounds will also need to be realigned (*c. f.* Section 2.3), using the same *alignment offsets* as in the fusion approach. However, the innermost loop body and the array indices remain essentially unchanged. From the cache-access point of view, both approaches, with and without fusion, are equivalent. Hence, we present our work based on the code template shown in Figure 2(b) without loop fusion, although the main analysis remains valid even if the inner loops are fused before tiling.

After tiling, the T-loop becomes the T'-loop in Figure 2(b). The new limits of the T'-loop and the loops within will be determined by formulas presented later in this section. We first define several terms concerning this new loop nest.

Definition 2.1. We call the $L''_{i,k}$ loops inside the T'-loop *tile-defining loops*. These loops are nested in the exact order as the $L_{i,k}$ loops prior to tiling. The iteration subspace of the $L''_{i,k}$ loops form a *loop tile*.

In effect, each of the original spatial loops is strip-mined, with a strip size B_k uniformly applied to all loops at level k, $1 \leq k \leq n$. Different loop levels, however, may use different strip sizes.

Definition 2.2. We call the trip count, B_k, of loop $L''_{i,k}$ the *tile size* at loop level k. (Note that B_k is equal for all i.) Collectively, we call $(B_1, B_2, \ldots, B_n)$ the *tile*

```
DO T = 1, ITMAX
  DO L_{1,1} = l_{1,1}, u_{1,1}
    ...
      DO L_{1,n} = l_{1,n}, u_{1,n}
        ...
        e_1(A(...))
        ...
      END DO /* L_{1,n} */
      DO L_{2,n} = l_{2,n}, u_{2,n}
        ...
        e_2(A(...))
        ...
      END DO /* L_{2,n} */
  END DO /* L_{1,1} */
  ...

  DO L_{m[1],1} = l_{m[1],1}, u_{m[1],1}
    ...
      DO L_{m[n]-1,n} = l_{m[n]-1,n}, u_{m[n]-1,n}
        ...
        e_3(A(...))
        ...
      END DO /* L_{m[n]-1,n} */
      DO L_{m[n],n} = l_{m[n],n}, u_{m[n],n}
        ...
        e_4(A(...))
        ...
      END DO /* L_{m[n],n} */
  END DO /* L_{m[1],1} */
  ...

END DO
```

(a) Before tiling

```
DO L'_1 = l'_1, u'_1, B_1
 ...
  DO L'_n = l'_n, u'_n, B_n
    DO T' = T'_low, T'_high

      DO L''_{1,1} = l''_{1,1}, u''_{1,1}
          ...
          DO L''_{1,n} = l''_{1,n}, u''_{1,n}
            ...
            e_1(A(...))
            ...
          END DO /* L''_{1,n} */
          DO L''_{2,n} = l''_{2,n}, u''_{2,n}
            ...
            e_2(A(...))
            ...
          END DO /* L''_{2,n} */
      END DO /* L''_{1,1} */
      ...

      DO L''_{m[1],1} = l''_{m[1],1}, u''_{m[1],1}
        ...
          DO L''_{m[n]-1,n} = l''_{m[n]-1,n}, u''_{m[n]-1,n}
            ...
            e_3(A(...))
            ...
          END DO /* L''_{m[n]-1,n} */
          DO L''_{m[n],n} = l''_{m[n],n}, u''_{m[n],n}
            ...
            e_4(A(...))
            ...
          END DO /* L''_{m[n],n} */
      END DO /* L''_{m[1],1} */
      ...

    END DO /* T' */
  END DO /* L'_n */
 ...
END DO /* L'_1 */
```

(b) After tiling

Fig. 2. Iterative stencil loops before and after tiling

size vector, or simply the *tile size* if no confusion results.

Definition 2.3. As T' increases from T'_{low} to T'_{high}, we say it completes a *tile traversal*. Two tiles are said to be *consecutive* within a tile traversal if the difference between the corresponding T' values is 1.

Definition 2.4. The T'-loop is enclosed in an n-deep perfect nest of loops, L'_1 through L'_n. We follow the terms in previous work [Wolf 1992] and call them the *tile-controlling loops*. These outer loops control the computation from one tile traversal to the next.

To illustrate the correspondence between the iteration spaces before and after tiling, take the example of the Jacobi kernel shown in Figure 1(a). To make the illustration simpler, we assume for the moment that the stencil loops are tiled at the J-level only. For the original stencil loops, we plot the product space of the T-iterations and the J-iterations (including both J_1 and J_2) as shown in Figure 3(a). Each dot represents either a pair (T, J_1) or a pair (T, J_2). Before tiling, the execution proceeds row by row. Within each row, the execution proceeds from the left to the right.

After tiling, a tile traversal covers the area of the iteration space between two stair-case lines in Figure 3(a), and a *tile* covers the shaded polygon. When the execution proceeds from one tile to the next in the same tile traversal, the tile boundaries for the J-loop are shifted to the left in order to satisfy data dependences. The single-level tile-controlling loop controls the computation from one tile-traversal to the next, in the direction from the top-left to the right-bottom. Notice that, unless $ITMAX = 1$, there exist more than $(N-2)/B$ tile traversals in the Jacobi example, where B is the width of the tile, i.e. the tile size. Also notice that the tile traversals are not necessarily of the same heights, due to skewing.

2.2 Data Dependences and the Skew Vector

A reordered execution sequence is legal only if it satisfies the original data dependences. We adopt a value-based definition of flow, anti- and output dependences [Wolfe 1995].

Definition 2.5. A *flow dependence* exists from statement $Stmt_1$ to statement $Stmt_2$ if the latter may use the value written by the former. An *anti-dependence* exists from $Stmt_1$ to $Stmt_2$ if the former may read a value before it is overwritten by the latter. An *output dependence* exists from $Stmt_1$ to $Stmt_2$ if $Stmt_2$ may overwrite a value written by $Stmt_1$.

One can similarly define dependences between two variable references, two loop iterations, two program segments and so on [Wolfe 1995]. For the Jacobi example, Figure 3(c) shows the flow dependences between the iteration points for $T = 1$ and 2, and Figure 3(d) shows the anti-dependences. (Output dependences are omitted from the figure.)

If a data dependence exists between two (T, J)-pairs, J being either J_1 or J_2, such that the J value of the dependence source is greater than that of the dependence sink, then we have a *negative dependence distance*. A dependence with a negative distance may exist within the same time step or between two different time steps. In Figures 3(c) and 3(d), the arrows pointing from right to left indicate

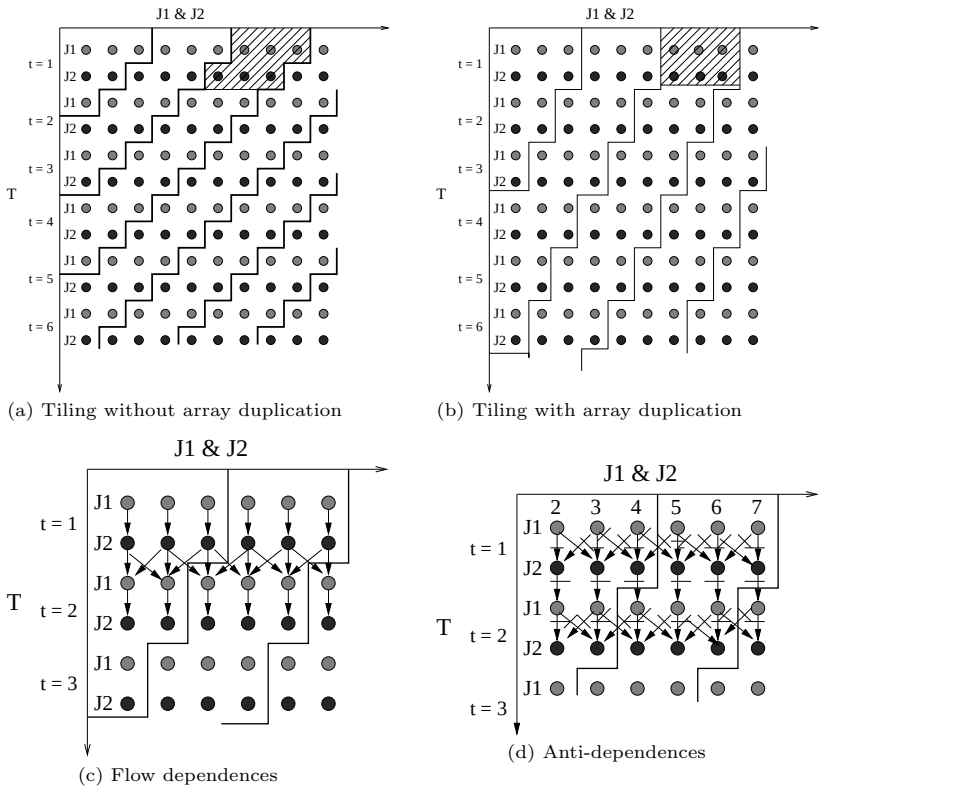


Fig. 3. Dependences and tiling of Jacobi

dependences with negative distances. In this example, the flow dependences with negative distances exist between consecutive time steps, and anti-dependences with negative distances exist within the same time step. It is easy to see that, due to the negative dependence distances, the tile traversals are not allowed to move straight down. (Otherwise, we will have a dependence source executing after the dependence sink.) Dependences with nonnegative distances do not impose this restriction on the tile traversals.

If no negative dependence distance exists, then all tile-defining loops at the same level can have identical loop limits. Consecutive tiles in the same tile traversal will then access identical data. In the presence of negative dependence distances, the loop limits need to be modified. First of all, the tile traversals must be *skewed*, as in Figure 3(a), such that no arrows will point from a later tile traversal to an earlier one. Suppose tiling is performed on n loop levels in the original loop nest, then the degrees of skewing at those loop levels are defined by a *skew vector* $\vec{S} = (S_1, \ldots, S_n)$, where S_k is called the *skew factor* at loop level k. In Figure 3(a), the tile traversal is skewed at the J level with the skew factor of 2. We shall give a formal definition for the skew vector. in the next subsection.

In the transformed loop nest, every time T' is increased by 1, the skew factor S_k will cause each tile-defining loop (at level k) to decrement both their lower and

upper limits by S_k. If dependences with negative distances exist between different time steps only, then each tile can retain the cuboidal shape. Otherwise, the tile-defining loop subnest which contains the sink of the dependence with a negative distance must be "left-shifted". That is, the lower and upper loop limits are both decremented by a vector of *alignment offsets*, or *offsets* in short. (A formal definition for the offsets is given in the next subsection.) As a result, the tile itself will also be skewed, as in the case of Figure 3(a), where the alignment offset is 1 for loop $J2$ and is 0 for loop $J1$. The next subsection defines the loop limits in the transformed loops (*c. f.* Figure 2(b)), given the skew factors, the offsets, and the tile sizes.

2.3 Determining the New Loop Limits

Tiling changes the loop structure, but it leaves the innermost loop body almost unchanged. Wherever the loop indices $L_{i,k}$ appear in Figure 2(a), they are changed to $L''_{i,k}$ in Figure 2(b). Other than that, the innermost loop body is unchanged after tiling. In the following, we present formulas to compute the loop limits in the transformed loops, given the tile size B_k at loop level k ($1 \leq k \leq n$).

2.3.1 *Formulas for the tile-defining loops*

Definition 2.6. Let the lower and upper limits of $L_{i,k}$ be $l_{i,k}$ and $u_{i,k}$ respectively. The lower limit of the corresponding tile-defining loop $L''_{i,k}$ equals $max(l_{i,k}, L'_k - (T'-1) * S_k - o_{i,k})$, and the upper limit equals $min(u_{i,k}, L'_k - (T'-1) * S_k + B_k - 1 - o_{i,k})$. The constant $o_{i,k}$ is called the *alignment offset*, or the *offset* in short, for each spatial loop $L_{i,k}$. The constant S_k is called the *skew factor* at loop level k, and $(S_1, S_2, \ldots, S_n)$ is called the *skew vector*.

There exist many legal choices for the skew vector and the offsets. An algorithm to determine the optimal skew vector and the corresponding offsets will be given in Section 4.

2.3.2 *Formulas for the tile-controlling loops.*

In the original loop nest, the loop limits $l_{i,k}$ and $u_{i,k}$ for each inner loop $L_{i,k}$ may vary as the value of T varies. Also, the loop limits for $L_{i,k}$ and $L_{j,k}$ may not be the same even though they are at the same loop level. Now, since the n-deep tile-controlling loops L'_k contain the T' loop, their loop limits must be T'-invariant. Furthermore, notice that each indexing vector of those tile-controlling loops marks the starting point of a tile traversal. In order to make sure that the tile traversals cover all iterations of the original spatial loops, the lower limit of each tile-controlling loop at level k should equal the lowest limit among all the spatial loops at level k. The upper limit should equal the highest limit at level k, plus an allowance resulting from skewing. (Without this allowance, the right-bottom corner of the original iteration space will not be fully covered, because of the skewing to the left.) At loop level k, a conservative estimate of the skewing allowance equals $S_k \times ITMAX - 1$, assuming that $L_{i,k}$ $(\forall i, \forall T)$ gets the highest upper limit when $T = ITMAX$.

Note that it is safe to overestimate the range of L'_k, because we have made the lower limits of the tile-defining loops no less than $l_{i,k}$ and the upper limits no greater than $u_{i,k}$. If the L'_k-loop index exceeds its proper range, then the tile-defining loops will become zero-trip loops, since their lower limits will be greater than their upper limits. Nonetheless, one can reduce the cost of loop-limit checking by tightening

the value range of L'_k. From the discussions above, we get a tight lower limit on L'_k equal to

$$\min\{l_{i,k} \mid \forall i \in [1, m[k]], \forall T \in [1, \text{ITMAX}]\}$$

and a tight upper limit equal to

$$\max\{u_{i,k} \mid \forall i \in [1, m[k]], \forall T \in [1, ITMAX]\} + S_k \times ITMAX - 1.$$

For most of the loop-limit expressions found in practice, the compiler can either determine the *min* and *max* values based on the above formulas, or it can insert operations into the program to compute these values efficiently at run time. We do not cover these details in this paper.

2.3.3 *Formula for the T' loop.* Because of loop skewing, some of the tile traversals may have heights that are less than $ITMAX$. (Take the Jacobi program as an example. In Figure 3(a), the leftmost tile traversal runs only two time steps, and its next tile traversal runs three time steps.) It is safe, however, to overestimate the range of T' as $[1, ITMAX]$. It is possible that, at run time, some of the T' values within this range will make the lower limit of a tile-defining loop greater than its upper limit. For example, consider a tile-defining loop $L''_{i,k}$ under $T' = ITMAX$, $L'_k = l_{i,k}$, $S_k = 1$ and $o_{i,k} = 0$. Based on the formulas in Section 2.3.1, this loop has the lower limit of $l_{i,k}$ and the upper limit of $l_{i,k} - ITMAX + B_k$. For $ITMAX > B_k$, the lower limit is clearly greater than the upper limit. In such cases, the tile-defining loop will make zero trips.

As with the tile-defining loops, we wish to tighten the range of T' in order to reduce the chances of executing zero trip tile-defining loops. For $L'_k \leq max\{u_{i,k} \mid \forall i, \forall T\}$, the T' loop has the lower limit of 1. For any index value of L'_k which is greater than $max\{u_{i,k} \mid \forall i, \forall T\}$, the tight lower limit on T' is equal to

$$\min\{\lfloor \frac{L'_k - \max\{u_{i,k} \mid \forall i, \forall T\}}{S_k} \rfloor \mid \forall k\}.$$

A general formula for the lower limit on T' can thus be written as

$$\max\{\min\{\lfloor \frac{L'_k - \max\{u_{i,k} \mid \forall i, \forall T\}}{S_k} \rfloor \mid \forall k\}, 1\}.$$

To tighten the upper limit, notice that the height of a tile traversal can be no greater than

$$\max\{\lceil \frac{L'_k + B_k - 1 - \min\{l_{i,k} \mid \forall i, \forall T\} - \min\{o_{i,k} \mid \forall i\}}{S_k} \rceil \mid \forall k\}.$$

In fact, this upper limit is reached for some index vectors, $(L'_1, L'_2, \ldots, L'_n)$, of the tile controlling loops. For example, in the very first tile traversal, we have $L'_k = \min\{l_{i,k} \mid \forall i, \forall T\}$, and we have the upper limit on T' equal to $\max\{\lceil \frac{B_k - 1 - \min\{o_{i,k} \mid \forall i\}}{S_k} \rceil \mid \forall k\}$. If we let T' take on this upper-limit value, then for some k and some i, T' equals $\lceil \frac{B_k - 1 - o_{i,k}}{S_k} \rceil$. For the same pair of k and i, the tile-defining loop $L''_{i,k}$ will have the upper limit of $L'_k - (T' - 1) * S_k + B_k - 1 - o_{i,k}$ which is greater than or equal to the lower limit $l_{i,k}$. This shows that at least one tile-defining loop has one or more iterations to execute under such a T'. Based on the discussion so far, we

derive a tight upper limit for T' as

$$\min\{\max\{\lceil \frac{L'_k + B_k - 1 - \min\{l_{i,k} \mid \forall i, \forall T\}}{S_k} \rceil \mid \forall k\}, ITMAX\}.$$

After tiling the Jacobi program, let the tile-defining loops be J''_1 (corresponding to J_1), I''_1 (corresponding to I_1), J''_2 (corresponding to J_2), and I''_2 (corresponding to I_2). Let the tile-controlling loops be J' (corresponding to J_1 and J_2) and I' (corresponding to I_1 and I_2). Recall that, in the original loop nest, the lower limits are 2 for all loops inside the T-loop. Suppose the skew factors are equal to 2 at both loop levels J and I, the offsets are equal to 0 for both J_1 and I_1, and the offsets are equal to 1 for both J_2 and I_2. Let B_I and B_J be the tile sizes yet to be determined at the I and J level, respectively. Substituting these numbers into the above formulas, we get the following loop limits after tiling.

—The lower limit of J''_1 equals $max(2, J' - (T' - 1) \times 2)$ and the upper limit equals $min(N - 1, J' - (T' - 1) \times 2 + B_J - 1)$.

—The lower limit of I''_1 equals $max(2, I' - (T' - 1) \times 2)$ and the upper limit equals $min(N - 1, I' - (T' - 1) \times 2 + B_I - 1)$.

—The lower limit of J''_2 equals $max(2, J' - (T' - 1) \times 2 - 1)$ and the upper limit equals $min(N - 1, J' - (T' - 1) \times 2 + B_J - 2)$.

—The lower limit of I''_2 equals $max(2, I' - (T' - 1) \times 2 - 1)$ and the upper limit equals $min(N - 1, I' - (T' - 1) \times 2 + B_I - 2)$.

—The lower limit of T' equals

$$\max\{\min\{\lfloor \frac{J' - N + 1}{2} \rfloor, \lfloor \frac{I' - N + 1}{2} \rfloor\}, 1\}$$

and the upper limit equals

$$\min\{\max\{\lceil \frac{J' + B_J - 3}{2} \rceil, \lceil \frac{I' + B_I - 3}{2} \rceil\}, ITMAX\}.$$

—The lower and upper limits of J' (also I') are equal to 2 and $N-1+2\times ITMAX-1$, respectively.

3. AN ANALYSIS OF MEMORY-REFERENCE COST

Intuitively, Figure 3(a) suggests that the further the tile traversals are skewed, the less overlap exists between the data referenced in consecutive tiles, and hence the sooner some previously cached data gets replaced. In this section, we give a memory-reference cost analysis which will show that the skew factors must be minimized in order to minimize cache misses. This analysis will also give the tile size which minimizes capacity misses. As in the rest of the paper, we focus on the memory references to array data and ignore the memory-reference cost to access instructions. For iterative stencil loops, the instruction references normally have a very high hit rate. Hence, their cost is negligible when compared to the cost to access the array data.

We consider a single processor which accesses caches at one or more levels. For simplicity of presentation, we limit our discussions to two levels of caches, namely the *primary cache* and the *secondary cache*. Generally speaking, the former is substantially smaller but faster than the latter. We assume both caches store data

only. This assumption is true at the primary cache level, because instructions and data normally have their own primary caches. A single secondary cache, however, may store both instructions and data. However, due to the high hit rate of the primary instruction cache when executing iterative stencil loops, the potential conflict between the instructions and the data in the secondary cache usually has little effect on the memory-reference cost.

Since the secondary cache has a much higher latency than the primary cache, the main objective of a tiling scheme is to minimize the primary-cache misses. It is worth pointing out, however, that there exist loops for which no tile sizes can make the array footprint fit in the primary cache [2]. In such cases, the objective is to minimize the secondary-cache misses. There is another factor which may force us not to target the primary cache. If the loop tile accesses too many different arrays, then the tile size will be very small due to the relatively small size of the primary cache. As a result, the innermost loop will have a low iteration count, making software pipelining ineffective. (Software pipelining is an important back-end compiler technique to exploit instruction-level parallelism between different loop iterations [Allan et al. 1995].) At the same time, little data reuse may be achieved on the primary cache due to the small iteration count. The break-even point for the tile size depends on the particular microprocessor and the particular back-end compiler. However, given a microprocessor and a back-end compiler, a general lower limit on the tile size can be easily obtained through experiments. We will discuss the limits we have used in our experiments in Section 7. In this section, we focus on analyzing the memory-reference cost.

3.1 Basic Terms and Assumptions

Let p_1 be the penalty for a primary cache miss and p_2 the penalty for a secondary cache miss. The total memory reference cost is simply

$$p_1 \times (\textit{No. of primary cache misses}) + p_2 \times (\textit{No. of secondary cache misses}). \quad (1)$$

For each cache, if the data size of the loop tile exceeds the cache size, then the number of cache misses is greater than or equal to

$$\frac{\textit{DataSize} \times \textit{ITMAX}}{\textit{CacheLineSize}}. \quad (2)$$

In the rest of this section, we give an estimate for the number of cache misses when the data size of the loop tile fits in the cache.

Cache misses can be divided into three classes, *compulsory misses*, *capacity misses* and *conflict misses* [Hennessy and Patterson 1996]. Compulsory misses occur when a memory block is referenced for the first time in a program's execution. Conflict misses are due to interferences between memory references whose addresses are mapped to the same cache line. *Self-interferences* exist between references to the same array (but different elements), and *cross-interferences* exist between references to different arrays [Lam et al. 1991]. All other cache misses are capacity misses.

[2]For example, data dependences may prevent the innermost loops from being tiled. The array footprint of such innermost loops, on the other hand, may exceed the primary cache size.

We wish to develop formulas to accumulate the total number of cache misses during the execution of the entire loop nest. Let the loop nest be tiled using the loop-tile size B_k and the skew factor S_k for each loop level k. We define our notation and state our assumptions below.

—All elements of an array which are referenced in a loop tile form an *array tile* [Panda et al. 1999]. If multiple arrays are referenced in a loop tile, then multiple array tiles exist. The total array footprint of the loop tile equals the union of its array tiles.
To make the discussions simpler, for a given loop tile size, all array tiles are assumed to have the same *shape*. This is usually true for the main arrays referenced in iterative stencil loops.

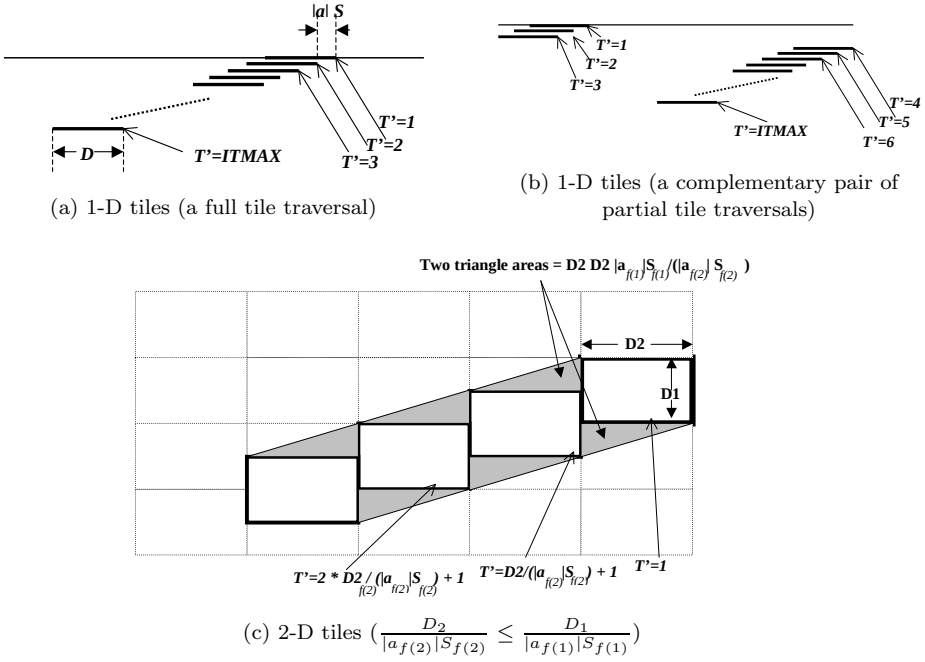
—Furthermore, since the differences among the array tile sizes are generally very small in iterative stencil loops, we assume the same size for the different array tiles.

—We assume that the array subscripts take the form $A(a_1 * I_1 + C_1, a_2 * I_2 + C_2, \ldots, a_m * I_m + C_m)$, where I_k is the index variable of a spatial loop and $a_k \neq 0$ is the *scaling coefficient* in the k-th dimension.

—We let $f(k)$ be the loop level at which the index variable runs through the k^{th} array dimension, and we assume $f(k)$ to be a one-to-one mapping. In the Jacobi example, we have $f(1) = 2$ and $f(2) = 1$. In Section 6.2.1, we discuss how to permute loops, according to *compatible loop levels*, such that spatial loops at the same level will have the index variables running through the same array dimension. Here, we assume that this loop permutation has been performed successfully. Since the array tiles are of the same shape, the scaling coefficient a_k in the k-th array dimension will be the same among all array references. The constant terms C_k, however, may be different for different array references.

—We let D_k be the *extent* of the array tile in the k-th array dimension. Clearly, D_k is equal to $|a_k|B_{f(k)}$ plus a constant term.

—We assume a uniform *stride*, g_k, between the neighboring elements in each dimension k of the array tile. Clearly, we have $1 \leq g_k \leq a_k$. If $g_k = 1$, then the array tile is fully packed in the k-th dimension. For example, if $A(2 * I)$ is the unique reference to array A in the iterative stencil loop, where I is the index variable of a tile-defining loop, then the array tile of A has a stride of 2. If, however, that same tile-defining loop contains yet another reference to array A, which is in the form of $A(2 * I + 1)$, then the array tile of A has unit stride, even though the scaling coefficient equals 2.
We assume g_1 to be no greater than the cache line size measured in the number of array elements which can be stored in a single cache line. Ideally, one wishes that $g_1 = 1$. If this condition is not met, then the given iterative stencil loop has poor spatial locality and should be transformed by a combination of loop permutation and array transpose [Kandemir et al. 1998] in order to minimize g_1. As a matter of fact, using a data transformation known as *compression* [O'Boyle and Knijnenburg 1997], one can make g_k equal to 1 for all k. For example, in the stride-2 example mentioned above, we can compress the array A by converting index $2 * I$ to I, which results in a unit stride. In this section, however, we do not



(a) 1-D tiles (a full tile traversal)

(b) 1-D tiles (a complementary pair of partial tile traversals)

(c) 2-D tiles ($\frac{D_2}{|a_{f(2)}|S_{f(2)}} \leq \frac{D_1}{|a_{f(1)}|S_{f(1)}}$)

Fig. 4. How many array tiles does a tile traversal cover?

assume unit stride when we develop formulas to estimate the memory reference cost.

If we make sure that the array footprint of each loop tile does not exceed the cache size, then no capacity misses will exist in any tile traversal. Furthermore, using techniques presented in Section 5, we can eliminate interferences between array references which occur within the same tile traversal. Thus, within each tile traversal, only the first reference to a memory block will incur a cache miss. (Note that the size of a memory block is equal to the size of a cache line.)

Conversely, we can reasonably assume that there exists no cache reuse between different tile traversals. For nonskewed tile traversals, this is due to the absence of data overlap between two different traversals. Some overlap may exist between two skewed tile traversals. However, the number of time steps, $ITMAX$, is generally large enough that, before the overlapping data can be reused in a later tile traversal, it has already been replaced in the cache. In summary, under the assumptions made above, *the number of cache misses in each tile traversal is approximately equal to the number of distinct memory blocks which are referenced.*

To derive the formulas that count cache misses, we first count the number of distinct memory blocks referenced in each *full* tile traversal. (In a full tile traversal, no tile-defining loops make zero trips for any $T' \in [1, ITMAX]$.) We then put all tile traversals together and find out how many *full* tile traversals they are equivalent to. We start with the simple case of one-dimensional array tiles. Afterwards, we work towards the arbitrary n-dimensional case.

3.2 The 1-D Case

Figure 4(a) illustrates the array tiles covered by a *full* tile traversal in the one-dimensional case. Assuming the skew factor of S and the scaling coefficient of a, each time T' increases by 1, the array tile shifts by $|a|S$ (to the left if $a > 0$ and to the right if $a < 0$). If $|a|S \leq D$, then a total of $(ITMAX\text{ - }1)|a|S/D + 1$ array tiles are covered by a full tile traversal. Suppose that there exist σ different arrays and that the array-tile stride equals g. The total number of distinct array elements referenced in a full tile traversal equals $((ITMAX\text{ - }1)|a|S/D + 1)\sigma D/g$. Dividing this number by $MemBlockSize/g$, where $MemBlockSize$ is the number of array elements which can be stored in each memory block, we get the number of distinct memory blocks referenced in each full tile traversal:

$$((ITMAX\text{ - }1)|a|S/D + 1)\sigma D/MemBlockSize. \tag{3}$$

If $|a|S > D$, then there exists no overlap between consecutive array tiles in the same tile traversal. A full tile traversal covers $ITMAX$ array tiles and the number of distinct memory blocks referenced equals $ITMAX \cdot \sigma D/MemBlockSize$.

Notice that a partial tile traversal either skips a lower range $T' \in [1, \bar{T}]$ (for some $\bar{T} < ITMAX$) or skips an upper range $T' \in [\bar{T}, \text{ITMAX}]$ (for some $\bar{T} > 1$). If $ITMAX \leq 1 + \frac{N-D}{|a|S}$, then there exists at least one full tile traversal. In this case, for each partial tile traversal which skips a lower T' range, $[1, \bar{T}]$, there exists a partial tile traversal which skips the complementary upper T' range, $[\bar{T} + 1, \text{ITMAX}]$, and *vice versa*. Such a pair of complementary partial traversals cover exactly the same number of array tiles as does a full tile traversal, as illustrated in Figure 4(b).

We add the number of full tile traversals to the number of partial tile traversals that skip the upper ranges. This sum should equal N/D, where N is the extent of the array accessed by the iterative stencil loops. Multiplying N/D by the number in Formula (3), we get the total number of cache misses for the case of $|a|S \leq D$:

$$((ITMAX\text{ - }1)|a|S/D + 1)\frac{DataSize \cdot g}{MemBlockSize}, \tag{4}$$

where $DataSize = \sigma N/g$ is the total number of distinct array elements referenced during all tile traversals. In the case of $|a|S > D$, the total number of cache misses in all tile traversals is equal to

$$\frac{ITMAX \cdot DataSize \cdot g}{MemBlockSize}. \tag{5}$$

We now consider the case in which $ITMAX$ is greater than $1 + \frac{N-D}{|a|S}$. In this case, no full tile traversals exist, and the partial tile traversals can be divided into three sets. The first set contains $(ITMAX\text{ - }1)|a|S/D + 1 - N/D$ tile traversals, each covering the full extent of the array. The second set contains N/D tile traversals, each covering a lower range only. The third set contains N/D tile traversals, each covering an upper range only. Putting the latter two sets of tile traversals together, we obtain N/D pairs of tile traversals, each covering the full extent of the array. Thus, the total number of cache misses in all tile traversals is still as shown in Formula (4) (if $|a|S \leq D$) or Formula (5) (if $|a|S > D$).

3.3 The n-D Case ($n \geq 2$)

Figure 4(c) illustrates how a tile traversal covers 2-D array tiles. The outermost box represents a two-dimensional array. A tile traversal sweeps through a part of the array in the direction determined by the signs of the scaling coefficients a_1 and a_2. Each time T' increases by 1, the array tile shifts by $|a_1|S_1$ in one array dimension and by $|a_2|S_2$ in another.

Let the array-tile size be $D_1 \times D_2$. Figure 4(c) is drawn with the arbitrary assumption that $\frac{D_2}{|a_{f(2)}|S_{f(2)}} \leq \frac{D_1}{|a_{f(1)}|S_{f(1)}}$. (If this inequality is reversed, the final formula will stay the same, even though it may change how the array tiles are aligned.) Recall that $f(i)$ is the loop level whose corresponding index variable runs through the array dimension i. After T' increases by $\frac{D_2}{|a_{f(2)}|S_{f(2)}}$, the array tile has shifted from the starting position to the left by the entire tile-width, D_2. Thus, the tile traversal covers an entirely new array tile *plus a pair of shaded triangles*. During this period of time, the array tile has also shifted vertically by the amount of $D_2 \cdot |a_{f(1)}| \cdot S_{f(1)}/(|a_{f(2)}|S_{f(2)})$. Therefore, the two shaded triangles have the total size of $D_2 D_2 |a_{f(1)}|S_{f(1)}/(|a_{f(2)}|S_{f(2)})$. The sum of the array *areas* (not the number of array tiles) covered by a full tile traversal is equal to

$$\frac{\mathit{ITMAX}\text{ - }1}{D_2/(|a_{f(2)}|S_{f(2)})} \left(D_1 D_2 + D_2 \frac{|a_{f(1)}|S_{f(1)} D_2}{|a_{(f(2)}|S_{f(2)}} \right) + D_1 D_2. \tag{6}$$

This number multiplied by $\frac{\sigma}{g_1 g_2} g_1/\mathit{MemBlockSize}$ is the number of distinct memory blocks referenced in a full tile traversal, which equals

$$(\frac{\mathit{ITMAX}\text{ - }1}{D_2/(|a_{f(2)}|S_{f(2)})}(D_1 D_2 + D_2 \frac{|a_{f(1)}|S_{f(1)} D_2}{|a_{(f(2)}|S_{f(2)}}) + D_1 D_2) \frac{\sigma}{g_2 \cdot \mathit{MemBlockSize}}. \tag{7}$$

As in the 1-D case, if there exists at least one full tile traversal in the 2-D case, then we can pair two complementary partial tile traversals such that the array area covered by the pair is of the same size as that covered by a full tile traversal. Let the full extent of the covered array be N_1 in one dimension and N_2 in the other. All partial and full tile traversals put together are equivalent to $\frac{N_1 N_2}{D_1 D_2}$ full tile traversals. Multiplying this expression by Formula (7), we get the total number of cache misses equal to

$$((\mathit{ITMAX}\text{ - }1)(\frac{|a_{f(1)}|S_{f(1)}}{D_1} + \frac{|a_{f(2)}|S_{f(2)}}{D_2}) + 1) \frac{\mathit{DataSize} \cdot g_1}{\mathit{MemBlockSize}}, \tag{8}$$

where $\mathit{DataSize} = \frac{\sigma N_1 N_2}{g_1 g_2}$. As in the 1-D case, it is not difficult to verify that the formula above applies even if there exist no full tile traversals. Finally, if D_1 (or D_2) is so much smaller than $|a_{f(1)}|S_{f(1)}$ (or $|a_{f(2)}|S_{f(2)}$) that there exist no overlap between consecutive array tiles in the same tile traversal, then the total number of cache misses in all tile traversals is determined by Formula (5), with g replaced by g_1.

Applying the same kind of reasoning to n-dimensional array tiles, we can write the formula for counting the total number of cache misses in all tile traversals as

$$((\Sigma_{k=1}^{n} \frac{|a_{f(k)}|S_{f(k)}}{D_k}) \cdot (\mathit{ITMAX}\text{ - }1) + 1) \frac{\mathit{DataSize} \cdot g_1}{\mathit{MemBlockSize}}, \tag{9}$$

unless there exists no overlap between consecutive array tiles, in which case Formula (5) applies (with g replaced by g_1). Here, *DataSize* equals to $\sigma\Pi_{j=1}^{n} N_j/g_j$.

Note that, if for any reasons the loop level $f(k)$ is not tiled, we should choose $D_k = N_k$ and $S_{f(k)} = 0$. From the loop construct shown in Figure 2, it is clear that all loops at the level $f(k)$ must be moved below the loop levels which are tiled. Also, if the tile-size selection scheme chooses $D_k = N_k$, it is easy to see that we can reduce cache misses by letting $S_{f(k)}$ be 0. No data dependences will be broken. This amounts to not tiling at loop level k.

From Formula (9) and Formula (5), we can immediately make the following claims.

Claim 3.1 *In order to minimize the memory reference cost, the skew factors must be minimized at all loop levels.* □

Claim 3.2 *Suppose that no interferences exist between array references within the same tile traversal and that none of the skew factors equal zero. To minimize the memory cost, the array-tile size D_k in each dimension k should be*

$$D_k = |a_{f(k)}|S_{f(k)} \cdot \sqrt[n]{\frac{\mathit{CacheSize}\ \Pi_{j=2}^{n} g_j}{\Pi_{j=1}^{n}|a_{f(j)}|S_{f(j)}}}. \tag{10}$$

Proof: Notice that to minimize the number of cache misses, we must let the product $D_1\Pi_{j=2}^{n}\frac{D_j}{g_j}$ be equal to *CacheSize*. Hence, we have $\Pi_{j=1}^{n}\frac{|a_{f(j)}|S_{f(j)}}{D_j} = \frac{\Pi_{j=1}^{n}|a_{f(j)}|S_{f(j)}}{\mathit{CacheSize}\ \Pi_{j=2}^{n} g_j}$. From Formula (9), clearly the number of cache misses is minimized if we let $|a_{f(j)}|S_{f(j)}/D_j$ be all equal (for all j), i.e.

$$(\frac{|a_{f(k)}|S_{f(k)}}{D_k})^n = \frac{\Pi_{j=1}^{n}|a_{f(j)}|S_{f(j)}}{\mathit{CacheSize}\ \Pi_{j=2}^{n} g_j}.$$

Equation (10) immediately follows.
□

If we have any $S_{f(k)} = 0$, Formula (9) shows that the factor $|a_{f(k)}|S_{f(k)}/D_k$ will have no impact on the memory reference cost. For nonzero $S_{f(k)}$, on the other hand, increasing the corresponding tile size D_k will decrease the memory reference cost. Therefore, for $S_{f(k)} = 0$, the corresponding tile size must be minimized so that the tile sizes in other dimensions can be maximized. For $k \neq 1$ such that $S_{f(k)} = 0$, we let $D_k = 1$. If $S_{f(1)} = 0$, we let $D_1 = \mathit{CacheLineSize}$. Among all nonzero $S_{f(k)}$, we should choose D_k such that the ratios of $|a_{f(k)}|S_{f(k)}/D_k$ are all equal.

4. MINIMIZING THE SKEW FACTORS

Based on Claim 3.1 in the last section, we wish to minimize the skew factors when we transform the given iterative stencil loops into the form in Figure 2(b). Data dependences in the given loops impose constraints on the minimum skew factors. Once the alignment offsets are fixed for the spatial loops, the minimum skew factors are further constrained. Hence, the correct approach is to compute the minimum skew factors allowed by the data dependences first, and to next find the corresponding alignment offsets.

In this section, we first give informal explanations for the constraints on the minimum skew factors. We then define an integer linear programming framework to determine the alignment offsets that minimize the skew factors. Finally, we recast this particular integer linear programming problem to a graph-theoretical one. Based on the minimum cost-to-time ratio cycle [Ahuja et al. 1993], we present a polynomial-time algorithm to find the minimum skew factors and the corresponding alignment offsets.

4.1 Effect of Data Dependence Distances

The minimum skew vector is constrained by data-dependence distances. In the current literature, dependence distances are usually defined with respect to loops which contain both of the dependent references. For our loop model (Figure 2(a)), this would apply to the T-loop only. We slightly extend the definition so as to also include the $L_{i,k}$ loops.

Definition 4.1. Given the nesting of loops shown in Figure 2(a), suppose two statements, A and B, are each embedded in n loops, where the two n-deep loop nests are not necessarily identical. Suppose there exists a dependence from statement A executed in iteration $\vec{i} = (t_i, L_{i,1}, \ldots, L_{i,n})$ to statement B executed in iteration $\vec{j} = (t_j, L_{j,1}, \ldots, L_{j,n})$. We say the dependence has a distance vector of $\vec{j} - \vec{i} = (t_j - t_i, L_{j,1} - L_{i,1}, \ldots, L_{j,n} - L_{i,n})$.

If $L_{j,k} - L_{i,k} < 0$, we say the dependence has a *negative distance* at the loop level k. For convenience, we use the term *backward distance*, at loop level k, to mean the absolute value of a negative distance, at loop level k. We call $t_j - t_i$ the T-distance of the dependence.

Allen and Kennedy proposed the terms of *loop-independent* versus *loop-carried* data dependences [Allen and Kennedy 1984]. Applying these terms to the above definition, a dependence is said to be *T-independent* if $t_j - t_i = 0$ and *T-carried* if $t_j - t_i > 0$.

Researchers have studied extensively the problem of computing the dependence distance between a pair of array references with respect to their common enclosing loops [Wolfe 1995]. When the distance is not constant, symbolic analysis can be performed to derive bounds on the distance values [Blume and Eigenmann 1998; Pugh 1992; Haghighat 1990]. This existing work can readily be applied to dependence distances as defined in Definition 4.1.

In Figure 3(c), the flow dependence edge from $J_1 = j_1$ to $J_2 = j_1$, where $2 \le j_1 \le N - 1$, is due to the write reference $temp(I_1, J_1)$ and the read reference $temp(I_2, J_2)$ within the same T iteration (*c.f.* Figure 1). Therefore, the corresponding dependence-distance vector is $(0, 0, 0)$. The flow dependence edge from $J_2 = j_2$ to $J_1 = j_2 - 1$, where $3 \le j_2 \le N - 1$, is due to the write reference $A(I_2, J_2)$ and the read reference $A(I_1, J_1 + 1)$ in two adjacent T iterations. Therefore, the corresponding dependence-distance vector is $(1, -1, 0)$. Similarly, we can compute the dependence-distance vectors for the other dependence edges in Figures 3(c) and 3(d).

In the following, we discuss the effect of data dependence distances on the alignment offsets and the skew factors.

4.1.1 *Alignment Offsets.* Recall that the purpose of using alignment offsets is to preserve the data dependences within the same time step. Therefore, the legality of an alignment offset is constrained by the T-independent data dependences only. Obviously, the alignment offsets at different loop levels are independent. On the other hand, alignment offsets at the same loop level are mutually dependent. Consider two distinct spatial loops i and j at level k and a T-independent dependence (with negative distance) from loop i to loop j. Relative to loop i, the tile boundaries for loop j should be "left-shifted" (i.e. decremented) at least by the amount of $-d_k$, where d_k is the dependence distance at level k. Hence, the alignment offset of loop j should be at least equal to the offset of loop i minus d_k. For a given nesting of iterative stencil loops, there exist many legal ways to assign alignment offsets to the loops at a given level k. In the following, we discuss the relationship between the alignment offsets and the skew factor at the same loop level.

4.1.2 *The Skew Factors.* The existence of T-carried dependences with negative distance affects the skewing of the tile traversals. The skew factors for different loop levels are independent. To analyze the relationship between the alignment offsets and the skew factor at the same loop level, we first examine an extreme case in which the alignment offsets are fixed as 0 for all loops at level k. Recall that if the skew factor, S_k, at loop level k is greater than zero, then every time T' is increased by 1, all tile-defining loops at level k will have their loop limits decremented by S_k. To satisfy a dependence whose distance is $d_k < 0$ at level k and $d_t > 0$ at the T-loop level, S_k should be no less than $-d_k/d_t$.

Now, to generalize the case, suppose that the offsets of loop i and loop j are not both 0 and that there exist a T-carried dependence from loop i to loop j with the T-distance d_t and the k-level distance d_k. Let the skew factor at the level k be S_k and the offsets of loops i and j be o_i and o_j, respectively. Within the time step $T = t$, the tile boundaries for loop j will be "left-shifted", relative to loop i, by $o_j - o_i$. In the time step $T = t + d_t$, the tile boundaries for loop j will be left-shifted by $o_j - o_i + d_t S_k$, relative to loop i in time step $T = t$. In order to satisfy this particular dependence, we should have $o_j - o_i + d_t S_k \geq -d_k$. Hence, the value of $(-d_k + o_i - o_j)/d_t$ is a lower bound on S_k.

Based on the discussion above, we can set up an integer linear programming framework which defines the minimum skew factor at loop level k.

Claim 4.1 *Given a nesting of iterative stencil loops (Figure 2(a)), consider all spatial loops at the level k and all dependences among such loops. Let $d_{i,j}$ denote the dependence distance from loop i to loop j at level k, and $t_{i,j}$ denote the T-distance. The minimum skew factor at loop level k should be equal to the minimum integer S which satisfies*

$$\begin{array}{l} o_j - o_i + d_{i,j} \geq 0, \ \ \forall T\text{-independent dependences } (i,j) \\ o_j - o_i + d_{i,j} + t_{i,j} S \geq 0, \ \ \forall T\text{-carried dependences } (i,j) \\ S \geq 0 \end{array} \tag{11}$$

where integers o_i and o_j denote the alignment offsets of loops i and j respectively. □

Since we have $t_{i,j} = 0$ for T-independent dependences, the inequalities in Claim 4.1

can be made more concise as in the following corollary.

Corollary 4.1 *The inequalities in Claim 4.1 are equivalent to*

$$\begin{array}{l} o_j - o_i + d_{i,j} + t_{i,j} S \geq 0, \ \ \forall \textit{dependences } (i,j) \\ S \geq 0 \end{array} \tag{12}$$

□

The problem defined in Claim 4.1 can be transformed into a graph-theoretical one, for which we can find a polynomial-time solution. For this purpose, we decompose a loop dependence graph $G = (V, E)$ [Wolfe 1995] by different loop levels.

Definition 4.2. The *loop dependence graph (LDG) at level* k is a graph (V, E) in which each node in V represents a spatial loop at level k (see Figure 2(a)) and each edge in E represents a dependence between two nodes in V. Multiple edges may exist from one node to another. For each edge, its *level-k* distance subvector (t, d) gives the T-distance t, and the distance d at the spatial loop level k.

For the Jacobi example, the symmetry between the loops at the I and the J levels causes the LDGs to look the same at both levels. Figure 5 shows the LDG for Jacobi.

The construction of the LDG requires information on array dataflow and dependence distances, which, in the worst case, requires exponential time to compute. However, in practice, such information can be obtained by efficiently implemented schemes (see [Gu et al. 1997; Wolfe 1995] for a list of references). Based on the LDG, we next develop a graph-theoretical solution for Ineq. (12).

4.2 A Graph-Theoretical Solution

Our graph-theoretical solution takes two steps. In the first step, we obtain a lower bound on the minimum skew factor that is independent of the alignment offsets. We prove that for such a lower bound S we can always find alignment offsets, o_i, that satisfy Ineq. (12). Thus, we find the minimum feasible solution of S for Ineq. (12). We derive the proof constructively by presenting an algorithm that, given such S, computes the alignment offsets o_i.

In an LDG, a path from node i to node j defines a transitive dependence from loop i to loop j. A simple cycle containing i defines a transitive dependence between two successive instances of loop i. Let $\sum t$ be the sum of the T-distances in a simple

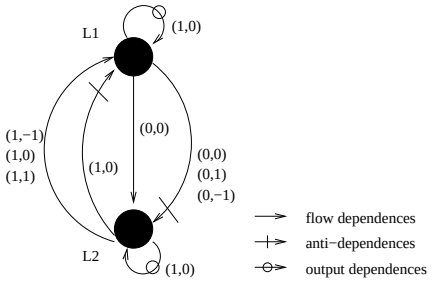


Fig. 5. The loop dependence graph of Jacobi

cycle and $\sum d$ be the sum of the k-distances. Clearly, the iteration x of loop i in the time step T must be executed before the iteration $x+\sum d$ of loop i in the time step $T+\sum t$. Hence, the value of $-\sum d/\sum t$ places a lower bound on the skew factor S at loop level k. Mathematically, this lower bound can be derived by summing up the inequalities in Ineq. (12) over the simple cycle, which gives $\sum d+\sum t\cdot S\geq 0$. Note that a simple cycle must contain at least one edge with $t>0$.

Let us take the maximum ceiling, S_{max}, of all such lower bounds collected from all simple cycles. S_{max} is obviously still a lower bound on the integer skew factor S, and is independent of the alignment offsets. If there exists an assignment of o_i that satisfies Ineq. (12) with $S=S_{max}$, then S_{max} is the minimum skew factor.

Next, we discuss how to compute S_{max} in polynomial time and then present a polynomial-time algorithm to compute the alignment offsets, given $S=S_{max}$.

4.2.1 *Computing S_{max}.* For each edge in E in the LDG whose level-k distance subvector is (t,d), we can define d as the cost of the edge and t the time. Then μ^* is the minimum cost-to-time ratio among all simple cycles in the given directed graph [Ahuja et al. 1993, §5.7], and $S_{max}=\lceil-\mu^*\rceil$. In the Jacobi example (Figure 5), there exist a simple cycle formed by two edges with dependence-distance subvectors $(1,-1)$ and $(0,-1)$, respectively. This simple cycle has the minimum cost-to-time ratio $\mu^*=(-1-1)/(1+0)=-2$. Thus $S_{max}=2$ is the minimum skew factor at both the I-loop and the J-loop levels.

Ahuja et al. [Ahuja et al. 1993, §5.7] present a binary search algorithm to find μ^*. They start with a range $[\underline{\mu},\overline{\mu}]$ to bound μ^* from below and from above, respectively. In our case, this range can be initialized to $[\sum_{d<0} d,\sum_{d>0} d]$. The value of μ^* is initially predicted to be $\mu^0=(\underline{\mu}+\overline{\mu})/2$. Using a shortest-path finding algorithm (in V^3 time), they show how to decide whether μ^0 overestimates or underestimates μ^*. If μ^0 is an overestimate, then it becomes the new upper bound for μ^*. If it is an underestimate, then it becomes the new lower bound for μ^*. If it is exactly equal to μ^*, then the minimum cost-to-time ratio and the simple cycle that produces it are both found. They prove that the exact μ^*, and the accompanying simple cycle, can be found by successively halving the range $[\underline{\mu},\overline{\mu}]$. The number of iterations taken to terminate such a binary search is in the order of $O(log(V*t_{max}*C_{max}))$ where, in our case, t_{max} is the maximum value of t and C_{max} is the maximum value of $-d$. Since the size of the problem is in the order of $O(E*(log(E)+log(C_{max})+log(t_{max})))$, which is the number of bits needed to represent the LDG, the algorithm to find μ^* has a polynomial time complexity, which is the problem size multiplied by V^3/E.

4.2.2 *Computing Alignment Offsets, given $S_{max}=\lceil-\mu^*\rceil$.* We start by assuming that the LDG has a single strongly connected component (SCC). We define the *length* of each edge (i,j) to be $d_{i,j}+t_{i,j}S$, where $S=\lceil-\mu^*\rceil$ and μ^* is the minimum cost-to-time ratio of the LDG.

Lemma 4.1 *The LDG contains no negative length cycles.*
Proof:

Otherwise we would find a cycle over which we have $\Sigma d+S\cdot\Sigma t<0$. This derives $\Sigma d/\Sigma t<-S$. Since $S=\lceil-\mu^*\rceil$, we can write $S=-\mu^*+\epsilon$, where $\epsilon\geq 0$. We then have

$$\mu^* = -S + \epsilon \geq -S > \Sigma d / \Sigma t,$$

which contradicts the fact that μ^* is the minimum cost-to-time ratio.
□

We pick an arbitrary node, say x, as the source of the given LDG and then define the *distance*, δ_i, of a node i to be the length of the shortest path from x to i. Using a label-correcting algorithm [Ahuja et al. 1993], we can determine δ_i for all i in $O(V \cdot E)$ time. We then assign the alignment offset $o_i = -\delta_i$ to each node i.

Lemma 4.2 *Let $o_i = -\delta_i$ for all i in the LDG, Ineq. (12) is satisfied.*
Proof:

For each edge (i, j), since δ_j is the shortest length of any path from x to j, we have $\delta_j \leq \delta_i + d_{i,j} + t_{i,j}S$, which immediately derives $o_j - o_i + d_{i,j} + t_{i,j}S \geq 0$,
□

For the Jacobi example (Figure 5), the LDG consists of a single SCC at both the I-loop and the J-loop levels. We arbitrarily pick the node $L1$ as the source. By definition, both the distance and the alignment offset of $L1$ are equal to 0. The shortest edge from $L1$ to $L2$ has length $d + t \cdot S = -1 + 0 \times 2 = -1$. Hence, we have $\delta_{L2} = -1$ and the alignment offset $o_{L2} = 1$ for node $L2$. It is easy to verify that Ineq. (12) is satisfied by all edges.

We now consider an arbitrary LDG which may contain more than one SCC. In $\Theta(V+E)$ time, we can construct the *component graph*, G_{SCC}, of the LDG [Cormen et al. 1990]. The graph G_{SCC} has one node for each SCC in the LDG. Suppose a and b in G_{SCC} represent two different SCCs, A and B, respectively. We draw an edge from a to b if there exist any edge in the LDG from a node in A to another node in B. We assign a *slack* to the edge (a, b) which is equal to

$$w(a, b) = max(-d_{i,j} - t_{i,j}S \,|\, \forall (i, j) \,\text{such that}\, i \in A,\, j \in B).$$

We first treat the SCCs independently and compute the alignment offsets as if each SCC is the only one in the LDG. Next, we use the slacks in G_{SCC} to adjust the alignment offsets between different SCCs. To do this, we recursively define a slack for each node in the acyclic directed graph G_{SCC}.

Definition 4.3. Suppose we have independently computed the alignment offsets for each SCC. For an SCC which is represented by a in G_{SCC}, let o_a denote the maximum alignment offset among all nodes in that SCC. We define a slack W_a according to the following rules:

(1) If a has no predecessors, then $W_a = 0$.

(2) Otherwise, we let $W_a = max(W_i + o_i + w(i, a) \,|\, \forall (i, a))$.

Obviously, we can determine the slacks of all nodes in G_{SCC} in linear time by following a topological sort. For each node a in G_{SCC}, we find all nodes in the LDG which belong to the SCC represented by a, and we increment the alignment offsets of all such nodes by the slack W_a. Thus, Ineq. (12) is satisfied.

5. REMOVING CACHE-SET CONFLICTS

In Section 3, we gave the optimal array tile size D_k at each loop level k (*c.f.* Claim 3.2). This result was obtained under the assumption that there exist no cache-set conflicts during each loop-tile traversal. In this section, we discuss how to perform minimum array padding to remove cache-set conflicts, given the array tile size. We first review address mapping between cache and memory then present the main results.

5.1 Set Conflicts

Set conflicts occur when different memory blocks are mapped to the same set in the cache. If the number of competing memory blocks exceed the set associativity, some blocks get replaced from the cache.

Suppose the cache has size C (in the number of cache lines) and associativity k. The cache is divided into C/k sets with indices from 0 to $C/k - 1$. Each cache set contains k cache lines. Virtual memory can be viewed as consisting of $\mathit{MemorySize}/\lambda$ memory blocks, where λ is the cache line size. A data word, x, at memory address MEM_x, has memory block index $\mathrm{BI} = \lfloor \mathrm{MEM}_x/\lambda \rfloor$ and is mapped to the cache set with the set index BI $mod\ C/k$.

For the Jacobi example, consider an extreme case in which each column of the arrays takes up storage equal to the cache size. Let the base address of array A be MEM_A and the word size be w. Under the column-major allocation scheme, assuming all arrays are aligned at cache line boundaries, array element $A(I_2, J_2)$ will have the memory address

$$\mathrm{MEM}_A + (J_2 - 1) \cdot C \cdot \lambda + (I_2 - 1) \cdot w,$$

and its set index in the cache will be

$$\lfloor (\mathrm{MEM}_A + (I_2 - 1)\, w)/\lambda \rfloor \quad (mod\ C/k).$$

It is easy to see that no matter what the column size, D_1, of the array tile is, all the elements of array A in the same tile row will be mapped to the same set in the cache. Self interferences will occur unless D_2 is no greater than k. If D_2 exceeds k by any integer value, d, there will be $D_1 \times d$ conflict misses due to self interferences within each tile of A. The same phenomenon can happen to the array *temp*.

There may also exist cross interferences between different arrays. In the Jacobi example, if the distaince between A and *temp* in the memory is a multiple of C, then the A elements and the *temp* elements which have the same row number will be mapped to the same cache set.

In less extreme cases, the choice of the array tile size affects how the array elements are mapped to the cache sets. For example, suppose the array column size in Jacobi equals $C + \lambda$. If the tile column size is chosen to be $D_1 = \lambda$, then two consecutive tile columns will be mapped to two consecutive cache sets without any conflicts. On the other hand, if the tile column size is chosen to be $D_1 = 2\lambda$, then two consecutive tile columns will have an overlap of one cache set. For a direct-mapped cache, this means that half of the array elements accessed in each tile column will be evicted from the cache before they get reused. Previous research efforts have attempted to select conflict-free array tile sizes. However, without

changing the array sizes, the conflict-free tile sizes generally do not agree with the optimum size determined in Claim 3.2. Therefore, a better approach is to keep the tile size at the computed optimum. At the same time, the cache set mapping can be altered by changing the extent of the array dimensions. This technique is known as *array padding*.

5.2 Our Array-Padding Scheme

Two kinds of array padding have been proposed to date [Bacon et al. 1994; Panda et al. 1999; Rivera and Tseng 1999]. *Intra-array padding* increases the extent of certain array dimensions to reposition the beginning address of each column (and each plane, and so on, for arrays of higher dimensions). *Inter-array padding* increases the distance between two different arrays in memory. For multilevel tiling in general, previous methods mainly rely on exhaustive search to find the correct pad size. In the following, we discuss how to find the pad size analytically, and therefore more efficiently, for tiled iterative stencil loops.

Our approach is to first analyze the conditions under which an m-dimensional array tile is free of self interferences and yet can fully utilize the cache. Given the tile size and the cache size, we determine the conditions which should be satisfied by the extent of each array dimension. We then find the minimum pad size to satisfy such conditions. Next, in order to minimize cross interferences between tiles from different arrays, we divide the cache among different arrays such that each array tile receives a sufficient number of cache sets. The starting addresses of these arrays are then adjusted, by inter-array padding, to minimize cross interferences.

We begin by assuming that the tiled loop accesses a single array, so that we have the whole cache for the single array tile. We will later handle multiple array tiles. We also assume that the uniform array-tile stride g_k equals 1 in each dimension k. How to handle $g_k > 1$ will become clear after we present the main theorem (Theorem 5.1). We consider an array of m dimensions, where $m > 1$. (Note that $m = 1$ is a trivial case, since a one-dimensional array tile consists of consecutive words. As long as the array tile fits in the cache, there will be no self interferences.)

Under the column-major allocation scheme, we number the array dimensions in the lexical order. The first dimension is the one that runs through consecutive words in the memory. Let N_i denote the size of the array in the i-th dimension, measured in words. The index in the i-th dimension ($i > 1$) runs through the memory in a stride of $\Pi_{j=1}^{i-1} N_j$.

For convenience, in the rest of the text, unless stated otherwise, array tile sizes and the cache size C will be measured in words. This is because only the first dimension of the tile is required to be a multiple of cache lines.

Let D_i be the size of the array tile in the i-th dimension. A maximum array section in the first dimension of the tile is called a *tile column*. A maximum two-dimensional section in the first two dimensions is called a *tile plane*. A maximum three-dimensional section in the first three dimensions is called a *tile cube*. A maximum d-dimensional section in the the first d dimensions is called a *tile d-cube*. Before continuing with the main discussion, we establish a few facts as follows.

Claim 5.1 *To fully utilize the cache and yet remain free of self interferences, the tile column size D_1 must divide C.*

Proof:. This claim is true because an array tile consists of $\Pi_{j=2}^{m} D_j$ tile columns, each of size D_1. □

Lemma 5.1 *Suppose a two-dimensional array tile has column size D_1 and row size D_2 such that D_1 is a multiple of the cache line size, D_1 divides C, and $D_1 \cdot D_2 = C$. This array tile can fully utilize the cache and remain free of self interferences if and only if D_1 divides N_1 and $GCD(N_1, C) = D_1$.*

Proof. The "if" part. Since $D_1 \times D_2 = C$, what remains to be shown is that no two distinct tile columns will ever share the same cache line. Let MEM_x be the starting memory address of the first tile column, and let two distinct tile columns have the beginning addresses $\mathrm{MEM}_x + a\,N_1$ and $\mathrm{MEM}_x + b\,N_1$, respectively, such that $0 \le a \le D_2 - 1$, $0 \le b \le D_2 - 1$, and $a \ne b$. Since D_1 divides both N_1 and C, $b\,N_1 - a\,N_1 (mod\ C)$ should be a multiple of D_1. We only need to prove that

$$b\,N_1 - a\,N_1 \neq 0 \pmod{C}. \tag{13}$$

If this inequality did not hold, then C/D_1 must divide $b\,N_1/D_1 - a\,N_1/D_1$. But $\mathrm{GCD}(N_1, C) = D_1$, so $C/D_1 = D_2$ must divide $b - a$, which is impossible unless $b = a$, because $|b - a| < D_2$. Hence Ineq. (13) must hold.

The "only-if" part. The starting addresses of two adjacent tile columns are N_1 words apart in the memory. Therefore they are $N_1\ (mod\ C)$ words apart in the cache. Hypothetically suppose D_1 does not divide N_1. Then D_1 does not divide $N_1\ mod\ C$ either, which leaves a gap between two tile columns in the cache and the gap is not a multiple of D_1. But from Claim 5.1, D_1 must divide C. Therefore, it is impossible to fully utilize the cache without creating self interferences. This shows that D_1 must divide N_1.

Next, we prove $\mathrm{GCD}(N_1, C) = D_1$, or equivalently, $\mathrm{GCD}(N_1/D_1,\ C/D_1) = 1$. We prove this by contradiction. Suppose

$$GCD(N_1/D_1, C/D_1) = \sigma > 1.$$

Let $N_1/D_1 = p\,\sigma$ and $D_2 = C/D_1 = q\,\sigma$. For any integer a, let $b = a + q$. We have $|b - a| = q < D_2$, but $N_1(b-a)/D_1 = 0\ (mod\quad C/D_1)$, i.e. $N_1(b-a) = 0\ (mod\ C)$, which is a contradiction.

□

Based on Lemma 5.1, we have the following array-padding algorithm for $m = 2$, where D_1 divides C. (Later, we discuss the case where D_1 does not divide C.) In all algorithms, we assume the cache size C is a power of two, as is the case in practice.

Algorithm 5.1 (Intra-array padding for a single two-dimensional array tile)

Let $N' = \lceil N/D_1 \rceil$. If N' is an odd number, then increase the array column size to $N'D_1$. If N' is an even number, then increase the array column size to $(N'+1)D_1$.

Since C/D_1 should remain a power of two, from Lemma 5.1, the cache will be fully utilized without self interferences.
□.
As an example, consider a single array with column size $N = 2^{10}$. Suppose we have the skew factors $S_1 = S_2 = 2$ and the loop index mappings $f(1) = 2$ and $f(2) = 1$. Formula 10 gives us $D_1 = D_2 = \sqrt{C}$. For $C = 2^{14}$ words, we have $D_1 = D_2 = 2^7 = 128$ words. $N' = \lceil N/D_1 \rceil = 8$ is an even number, so we increase the array column size to $(N' + 1)D_1 = 1152$.

We now consider array tiles of dimensions higher than two. To develop an intuition, consider $m = 3$ first, where $D_1 \times D_2 \times D_3 = C$. We first use Algorithm 5.1 to choose a new column size N_1 for the array, such that $\mathrm{GCD}(C, N_1) = D_1$. Next, we want to choose a new N_2.

We can view the cache as being divided into C/D_1 chunks, each of size D_1. We number these chunks from 0 to $C/D_1 - 1$ and call these numbers the *chunk indices*. Each tile plane (of size $D_1 \times D_2$) is mapped to D_2 chunks. Although these chunks are not necessarily adjacent in the cache, their chunk indices will be a consecutive subsequence of $\{0, 1, 2, \ldots, C/D_1 - 1\}$. Thus, just as a tile column is mapped to a series of consecutive cache sets in the case of $m = 2$, a tile plane is mapped to a series of consecutive chunk indices in the case of $m = 3$. What we want to find is a new N_2 such that, for any pair of distinct tile planes, the corresponding pair of chunk-index subsequences will have no overlap. Following the same steps in the proof of Lemma 5.1, we easily get the following lemma.

Lemma 5.2 *Suppose a three-dimensional array tile has the size of D_1, D_2 and D_3 in respective dimensions, such that $D_1 \times D_2 \times D_3 = C$. This array tile can fully utilize the cache and remain free of self interferences if and only if the following conditions are met: (1) D_1 divides N_1; (2) $GCD(N_1, C) = D_1$; (3) D_2 divides N_2; and (4) $GCD(N_2, C/D_1) = D_2$.*
□
We can apply the same idea inductively to the general case of $m > 2$. This leads to the following theorem.

Theorem 5.1 *Suppose an m-dimensional array tile has the size of D_i in the i-th dimension, such that $D_1 \times D_2 \times \ldots \times D_m = C$. This array tile can fully utilize the cache and remain free of self interferences if and only if the following conditions are met:*

(1) D_i divides N_i;

(2) $GCD(N_1, C) = D_1$; and

(3) $GCD(N_i, C/(D_1 \times D_2 \times \ldots \times D_{i-1})) = D_i$.

□
Based on the theorem above, Algorithm 5.1 can be extended, in a straightforward manner, to the general $m > 2$ case.

We now discuss how to deal with the case of the array-tile stride $g_k > 1$. Recall that we assume k_1 to be no greater than the cache line size λ (*c. f.* Section 3.1). In order to utilize all the cache lines, we should have $C = D_1 \Pi_{j=2}^{n} D_j/g_j$. Hence, both D_1 and D_k/g_k, for all $k > 1$, should divide C. We can still use Algorithm 5.1 to

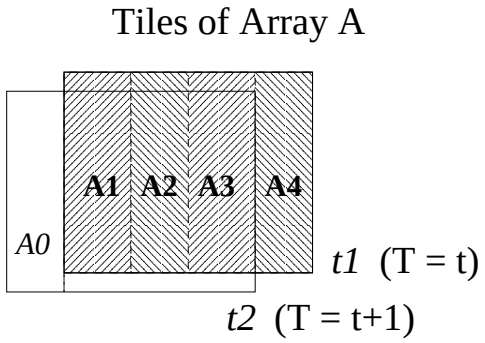


Fig. 6. Partial overlap of consecutive tiles

compute the new array size in the *first* dimension. For dimensions $k > 1$, however, we can view the "effective" array-tile size in the k-th dimension as D_k/g_k and modify Lemma 5.2 as follows.

Notice that there exists a gap of $N_2 \times g_2$ between the starting memory addresses of two neighboring tile columns. We need to modify condition (3) in Lemma 5.2 to "D_2/g_2 divides $N_2\, g_2$" and condition (4) to "$GCD(N_2\, g_2, C/D_1) = D_2/g_2$". The latter condition implies

$$GCD(N_2, \frac{C}{D_1\, \psi}) = \frac{D_2}{g_2\, \psi},$$

where $\psi = GCD(g_2, C/D_1)$. To compute the new array size in the second dimension, therefore, we compute $N_2' = \lceil N_2/(\frac{D_2}{g_2\, \psi}) \rceil$ which equals $\lceil N_2\, g_2\, \psi/D_2 \rceil$. If N_2' is an odd number, then the new array size in the second dimension should equal $N_2' D_2/(g_2\, \psi)$. Otherwise, the new array size should equal $(N_2' + 1)D_2/(g_2\, \psi)$. Theorem 5.1 should be modified in a similar way to compute the new array size in higher dimensions.

Next, we discuss three remaining issues: how to remove self interferences between the overlapping tiles (belonging to the same array) which are covered in the same tile traversal, how to remove cross interferences between tiles belonging to different arrays, and how to treat tile sizes which do not divide the cache size.

5.2.1 *Interferences due to Skewing.* Previous tile-size selection schemes target nonskewed tiling. For skewed tiling, we need to deal with a special kind of conflict miss that was not discussed before. Here, the loop limits of a tile-defining loop decrease as the time step increases. Thus, two consecutive loop tiles (and the corresponding array tiles) overlap only partially. Figure 6 shows two consecutive array A tiles *t1* (for $T' = t$) and *t2* (for $T' = t + 1$). Each tile consists of sections of four array columns. Suppose the array footprint of *t1* covers the whole cache, and the tile size has been chosen to remove self interferences from each tile. Without loss of generality, suppose each tile is swept through from left to right when the loop iterations are executed. The rightmost array section (marked as $A4$) will be the most recently referenced immediately before array tile *t2* is swept through. If the cache is two-way set-associative, then *A2* and *A4* will map to the same cache sets, $A4$ being the least recently referenced. The left-most array section in *t2* (marked

as $A0$) will then replace $A2$. However, a large part of $A2$ could have been reused.

The interferences between consecutive tiles as illustrated above can occur on set-associative caches, but not on direct-mapped caches. To remove such interferences, we can view a pair of consecutive tiles as a single tile and try to eliminate self interferences from the combined tile. Thus, after deriving a tile size which is free of self interferences within a single tile, we simply strip off $|a_{f(j)}|S_{f(j)}$ columns (or rows) in each dimension j of the array tile. The reduced tile size will guarantee that there exist no interferences between two consecutive tiles.

5.2.2 *Multiple Arrays.* Next, we consider multiple arrays. We assume the array tiles for different arrays to be of the same shape and size, as is often the case in iterative stencil loops. Suppose we have n array tiles, each of size $D = D_1 \times D_2 \times \ldots \times D_m$ such that $n \times D$ equals the cache size. We divide the cache into n equal parts and assign one part to each array tile. (If n is not a power of two, we round it up to the next power of two.)

We first perform intra-array padding on each array. In the cache, we want to leave a gap between two consecutive tile columns (from the same array) which has just enough cache locations for $n-1$ tile columns, each from a distinct array. Thus, when we apply Algorithm 5.1 and its higher-dimensional extensions to perform intra-array padding, we do so as if the tile column size is nD_1, instead of D_1. Furthermore, the cache size used in the algorithm remains C, which allows each array tile to be mapped to C/n different words in the cache.

After intra-array padding is performed on all of the n arrays, we number them from 1 to n. We then perform inter-array padding such that the starting memory addresses of the i-th and the $i+1$-th arrays are D_1 words apart, modulo C.

5.2.3 *Applying These Techniques to the Jacobi Example.* We can now illustrate the process of tile-size selection and array padding using the Jacobi example, performing tiling without array duplication. Assume the cache size is $C = 2^{15}$ words, the cache line size is $\lambda = 8$ words, and the array size is $N \times N = 1200 \times 1200$ words. Dividing C between the two arrays A and *temp* gives each array 2^{14} words in the cache. Plugging Cachesize $= 2^{14}$, $S_1 = S_2 = 2$, $f(1) = 2$ and $f(2) = 1$ into Formula 10, we get $D_1 = D_2 = 128$ words for each array tile. Since we have two arrays here, we pretend that the tile column size is $2D_1 = 256$ words when we apply Algorithm 5.1. We get $N' = \lceil N/256 \rceil = 5$ in return. Since N' is an odd number, we increase the array column size to $N' \times 256 = 1280$. If A and *temp* are allocated immediately adjacent to each other in memory, their starting addresses will be $1280 \times 1200 = 153600$ words apart, which equals 22528 (modulo C). To make their starting addresses $D_1 = 128$ words apart (modulo C) instead, We insert between A and *temp* a padding array of size equal to $C - 22528 + D_1 = 10368$ words. Finally, in order to remove interferences between two consecutive tiles in the same tile traversal, we remove two columns and two rows from each array tile, reducing the array tile size to $D_1 = D_2 = 126$ words. The loop tile size at both loop levels should be $B_J \times B_I = 124 \times 124$, because each loop tile accesses $B_I + 2$ tile rows and $B_J + 2$ tile columns in array A.

5.2.4 *Treating Tile Sizes Which Do Not Divide C.* Algorithm 5.1 and its higher-dimensional extensions require that the tile size in each dimension divides C. (In

the case of $g_k > 1$, for $k > 1$, the "effective" tile size D_k/g_k is required to divide C.) For the test programs we have experimented with so far, the tile sizes computed using Algorithm 5.1 often do divide C. However, for some test programs, the computed tile sizes do not divide C. In such cases, we round the tile size (or the "effective" tile size) in each dimension to the nearest power of two, making sure that $\Pi_{j=1}^{n} D_j = C$ (or $D_1 \Pi_{j=2}^{n} D_j/g_j = C$). Algorithm 5.1 and its extensions can then be applied.

6. ENHANCEMENTS

In this section, we describe several enhancements to the main tiling algorithm presented in Section 4. We first present three techniques to reduce the minimum skew factors. One is based on *array duplication* and the other two are based on loop transformations. We also present a scheme to tile iterative stencil loops which may terminate before the T-loop executes all of the $ITMAX$ iterations. Such early termination is normally due to the existence of a convergence condition which may be met before T reaches $ITMAX$.

6.1 Array Duplication

The analysis in Section 3 shows that the memory reference cost of iterative stencil loops is proportional to the skew factors. From the discussion in Section 4, we know that the minimum skew factor at any loop level is equal to the negation of the minimum cost-to-time ratio, $\Sigma d/\Sigma t$, among simple cycles in the loop dependence graph at that loop level. One way to increase the cost-to-time ratio is to reduce the negative dependence distances between the spatial loops. The other way is to increase the time, i.e. the T-distance. In this subsection, we discuss array duplication as a method to increase the T-distance of anti-dependences and output dependences.

It is known that T-independent output dependences can be removed by static *array renaming* [Wolfe 1995], because the dependent references write different values. If two array references do not have any flow dependences, then any anti-dependences between them can also be completely removed by static *array renaming*. This is because two such references share the same addresses, but not the same data values. However, if two references have both flow and anti-dependences, then the anti-dependences cannot be completely removed, since the two references share the same values. The Jacobi program is a good example. The read references of A in loop nest L_1 and the write references of A in loop nest L_2 share values. Thus, the anti-dependences from L_1 to L_2 on array A cannot be completely removed.

For array references that share the same values, we use a special kind of array renaming, called *array duplication*, to change T-independent anti-dependences into T-carried ones. A special case of array duplication is *odd-even duplication*, in which only one duplicate is created. The duplicated array has two versions, one storing the values written in the odd T-iterations, and the other storing the values written in the even iterations. In this way, the T-distance, i.e. the time, of these dependence edges is increased from 0 to 1. Creating m duplicates for the same array will increase the T-distance to m, and the values will be written to the $m + 1$ versions of the array in a round-robin manner. We should point out that array duplication is feasible only if we can correctly reconstruct the data flow after duplication. If,

for any reason we cannot reconstruct the data flow, e.g. if the *flow* dependence distance is uncertain for the target array due to some IF conditions, then we do not duplicate the array.

Since array duplication increases the working set of the loop nest, we need to strike a balance between the benefit of reduced skewing and the potential penalty of the increased working set. In general, suppose duplicating an array can increase Σt by a factor of δ_t, where Σt is the denominator in the minimum cost-to-time ratio, but at the expense of increasing the data size by a factor of δ_{size}. The skew factor S_k is then approximately reduced by a factor of δ_t. The reason this is an approximate estimate is because the skew factor should be rounded to the nearest integer. For example, if $0 < \Sigma d/\Sigma t < 1$, the skew factor becomes 1 and can no longer be decreased any further by increasing t. According to Formula (9) in Section 3, the memory-reference cost becomes δ_{size}/δ_t times the previous memory-reference cost. Clearly, array duplication is beneficial only if $\delta_{size} < \delta_t$.

Take the Jacobi program (Figure 1(a)) for example. Under odd-even duplication, the Jacobi program can be transformed as shown in Figure 7(a), where a duplicate A' is made for the array A. (Note that the resulting code is shown under the assumption that $ITMAX$ is an even number. If $ITMAX$ is an odd number, then A' will be written in the even T'-iterations, so that at the end of the execution, the original array A will store the live values.) The array duplication increases the T-distance of the anti-dependence on array A from 0 to 1. As a result, the total time of the simple cycle $L1 \rightsquigarrow L2 \rightsquigarrow L1$ is increased from 1 to 2, while the total cost remains the same. Thus, the skew factor is decreased from 2 to 1 at both levels of the spatial loops, I and J. On the other hand, the data size is increased to 3/2 times the original data size. Referring to Formula 9, we can see that the memory-reference cost is reduced to $(3/2)/(2/1) = 3/4$ times the previous cost. Figures 3(b-d) illustrate the new tiling at the J-loop level. As we shall see below, using the *forward substitution* technique [Wolfe 1995], we can reduce the data size back to two arrays and thus reduce the memory-reference cost by a factor of 2.

6.1.1 *Forward Substitution.* The technique of forward substitution substitutes a variable use by the right-hand side expression in the statement that defines its value. Before array duplication, a right-hand side expression may contain array references whose forward substitution is prevented by T-independent anti-dependences. If such anti-dependences are removed by array duplication, forward substitution may become legal. It is known that forward substitution often results in the substituted variables being dead. Such dead variables can be removed from the iterative stencil loops to reduce the working set.

In the Jacobi example, suppose that the array *temp* value is dead after the iterative stencil loops terminate. Prior to array duplication, it is illegal to substitute the right-hand side expression of statement "$temp(.) = \ldots$" into the use of *temp* in loop nest L_2, because doing so would violate the T-independent anti-dependences between the read references of array A in loop nest L_1 and the write references of A in loop nest L_2. After the duplication of array A, the T-independent anti-dependences disappear, which allows the forward substitution and the consequent dead-code removal as shown in Figure 7(b). Since the data size is the same before and after odd-even duplication followed by forward substitution, the memory cost is

```
      DO I = 1, N /* boundaries */
        A'(I, 1) = A(I, 1)
        A'(I, N) = A(I, N)
      END DO
      DO I = 2, N − 1
        A'(1, I) = A(1, I)
        A'(N, I) = A(N, I)
      END DO
      DO T = 1, ITMAX
L1 :    DO J1 = 2, N − 1
          DO I1 = 2, N − 1
            IF (MOD(T, 2).EQ.1) THEN
              temp(I1, J1) = (A(I1 + 1, J1)
                + A(I1 − 1, J1) + A(I1, J1 + 1)
                + A(I1, J1 − 1))/4
            ELSE
              temp(I1, J1) = (A'(I1 + 1, J1)
                + A'(I1 − 1, J1) + A'(I1, J1 + 1)
                + A'(I1, J1 − 1))/4
            END IF
          END DO
        END DO
L2 :    DO J2 = 2, N − 1
          DO I2 = 2, N − 1
            IF (MOD(T, 2).EQ.1) THEN
              A'(I2, J2) = temp(I2, J2)
            ELSE
              A(I2, J2) = temp(I2, J2)
            END IF
          END DO
        END DO
      END DO
```

(a) Jocobi after array duplication

```
      DO I = 1, N /* boundaries */
        A'(I, 1) = A(I, 1)
        A'(I, N) = A(I, N)
      END DO
      DO I = 2, N − 1
        A'(1, I) = A(1, I)
        A'(N, I) = A(N, I)
      END DO
      DO T = 1, ITMAX
L2 :    DO J2 = 2, N − 1
          DO I2 = 2, N − 1
            IF (MOD(T, 2).EQ.1) THEN
              A'(I2, J2) = (A(I2 + 1, J2)
                + A(I2 − 1, J2) + A(I2, J2 + 1)
                + A(I2, J2 − 1))/4
            ELSE
              A(I2, J2) = (A'(I2 + 1, J2)
                + A'(I2 − 1, J2) + A'(I2, J2 + 1)
                + A'(I2, J2 − 1))/4
            END IF
          END DO
        END DO
      END DO
```

(b) After forward substitution and dead code removal

Fig. 7. Jacobi transformed by array duplication

thus reduced by a factor of 2. Introducing m additional duplicates would be useless, as the memory cost will always be reduced by a factor of $(2+m)/((2+m)/2) = 2$.

After applying the transformations mentioned above, we have the following loop limits for the tiled Jacobi, based on the formulas in Section 2.3.

—The lower limit of J_2'' equals $max(2, J' - (T' - 1))$ and the upper limit equals $min(N - 1, J' - (T' - 1) + B_J - 1)$.

—The lower limit of I_2'' equals $max(2, I' - (T' - 1))$ and the upper limit equals $min(N - 1, I' - (T' - 1) + B_I - 1)$.

—The lower limit of T' equals $\max\{\min\{J' - N + 1, I' - N + 1\}, 1\}$ and the upper limit equals $\min\{\max\{J' + B_J - 3, I' + B_I - 3\}, \text{ITMAX}\}$.

6.1.2 *Identifying Array Duplication Candidates.* As discussed in Section 4, the binary search algorithm to find the minimum cost-to-time ratio also finds the simple cycle which produces the minimum ratio. For each anti-dependence, if any, in the cycle, we evaluate the memory-cost reduction factor δ_t/δ_{size} as the result of duplicating the referenced array. Only if the reduction factor is greater than one do we mark the anti-dependence as a candidate. The LDG is modified by revising the T-distances of all the marked anti-dependences.

We then recompute the minimum cost-to-time ratio, μ^*. If that ratio increases, we confirm the decision to remove those previously marked anti-dependences by array duplication. Otherwise, we have found another simple cycle which produces

```
DO T = 1, ITMAX
  DO J1 = 1, N
  DO I1 = 1, N
    ... ← A(I1, J1)
  END DO
  END DO
  DO J2 = 1, N
  DO I2 = 1, N
    A(J2, I2) ← ...
  END DO
  END DO
END DO
```

(a)

```
DO T = 1, ITMAX
  DO J1 = 1, N
  DO I1 = 1, N
    ... ← A(I1, J1)
  END DO
  END DO
  DO I2 = 1, N
  DO J2 = 1, N
    A(J2, I2) ← ...
  END DO
  END DO
END DO
```

(b)

```
DO T = 1, ITMAX
  DO I1 = 1, N
  DO J1 = 1, N
    ... ← A(I1, J1)
  END DO
  END DO
  DO J2 = 1, N
  DO I2 = 1, N
    A(J2, I2) ← ...
  END DO
  END DO
END DO
```

(c)

Fig. 8. Examples of loop compatibility

a μ^* value equal to the old one. Regardless of whether μ^* gets increased or not, we traverse the newly discovered simple cycle which has the minimum cost-to-time ratio in an attempt to find new candidates for array duplication. This continues until the newly discovered simple cycle (which has the minimum cost-to-time ratio) contains no duplication candidates. Since each traversal will mark at least one anti-dependence, it takes E traversals until the search terminates.

6.2 Reducing Backward Dependence Distances

In this subsection, we explore opportunities to reduce the minimum skew factors through loop transformations. We use such transformations to shorten backward dependence distances between spatial loops. We examine two methods for this purpose. One is called *compatible loop-level recognition*, and the other is called *circular loop skewing*.

6.2.1 *Compatible Loop-Level Recognition.* In Section 3.1, we made the assumption that the array subscripts take the form $A(a_1 * I_1 + C_1, a_2 * I_2 + C_2, \ldots, a_m * I_m + C_m)$, where I_k is the index variable of one of the spatial loops and a_k is the *scaling coefficient* in the k-th dimension. In a given nesting of iterative stencil loops, if the index variables at some loop level run through different dimensions of the same array, cache-miss estimation formulas in that section will be inaccurate. More importantly, the mismatched index may result in a long backward dependence distance.

Take the example in Figure 8(a). There exists a flow dependence from J_2 to J_1 with a distance vector $(1, 1-N, N-1)$, which yields a large skew factor, $N-1$, at the J level. This large skew factor results from the *incompatibility* between loops J_1 and J_2, in the sense that their index variables appear in different dimensions of array A. On the other hand, we say that loops J_1 and I_2 are *compatible* because their index variables appear in the same dimension of array A. Similarly, loop I_1 is said to be compatible with loop J_2.

If we permute loops J_2 and I_2 as shown in Figure 8(b), then the flow dependence from I_2 to J_1 has the distance vector $(1, 0, 0)$, making it more profitable to tile the loop nest. Similarly, loops I_1 and J_1 can be permuted as shown in Figure 8(c) to make loops compatible. However, assuming column-major allocation, spatial locality will suffer in this permuted loop nest. To restore the spatial locality, array A needs to be transposed [Anderson et al. 1995; Kandemir et al. 1998].

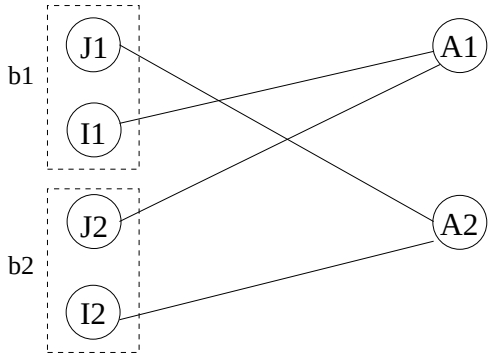


Fig. 9. An orientation graph

It is important to verify that all loop levels being considered for tiling are compatible. If they are found to be incompatible, the loop nesting order can be permuted, making them compatible. For this purpose, we build an *interference graph* in which each node represents an individual loop embedded in the T-loop. If two loops are incompatible, then an edge is drawn between the corresponding pair of nodes. Our task is to color the interference graph so that no adjacent nodes have the same color. Nodes of the same color will be given the same loop nesting level. If one spatial loop is embedded in another, they are required to have two different colors, and an edge should be drawn between the pair of loops.

The construction process of the interference graph is quite simple. We first build a graph called the *orientation graph* to associate loops with array dimensions. (Previously orientation graphs have been used for parallel loop scheduling and data allocation [Nguyen and Li 1998].) The orientation graph is a bipartite graph. The nodes on one side represent the individual loops, and the nodes on the other side represent the array dimensions. An edge is drawn between a loop node and an array-dimension node if the corresponding loop-index variable appears in the particular array dimension. Under the assumptions in Section 3.1, each loop node has one connecting edge only.

Figure 9 shows the orientation graph for the code example listed in Figure 8(a). In that graph, the nodes *A1* and *A2* represent the first and the second dimensions, respectively, of the array A. The loop nodes within each dashed-line box (b1 or b2) belong to the same perfect loop subnest. The construction of the orientation graph takes linear time in the size of the T-loop body. To build the interference graph, we go through each loop node L in the orientation graph: for each edge that connects L to an array-dimension node, we declare all loop nodes which connect to *other* dimensions of the same array to be interfering with L. As mentioned above, if one spatial loop is embedded in another, they interfere with each other. The time to construct the interference graph is proportional to its size, which, in the worst case, is the square of the number of loop nodes.

Figure 10(a) shows the interference graph for the code example shown in Figure 8(a). It is easy to see that J_1 and I_2 will have the same color and that J_2 and I_1 will have the same color. To take advantage of cache-line spatial locality, J_2 and I_1 are assigned to the innermost loop level. Figure 10(b) shows the interference

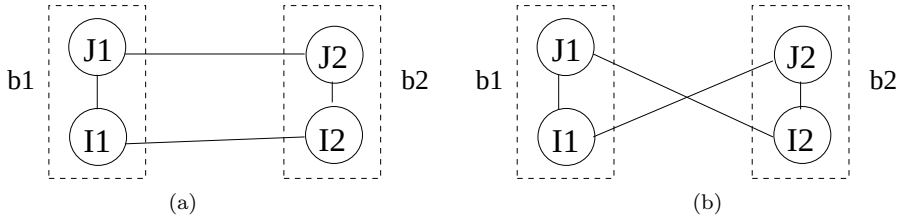


Fig. 10. Interference graphs

graph for the Jacobi code. It is clear that the original loop nesting order is good.

The problem of deciding whether a graph is n-colorable, $n \geq 3$, is known to be NP-complete [Gary and Johnson 1979]. However, there exist good heuristic methods which take time linear in the size of the interference graph. We use a method which sequentially removes the nodes from the graph and places them in a coloring stack. In each step, the node with the fewest neighbors is removed [Briggs et al. 1989; Matula and Beck 1981]. When the graph becomes empty, the nodes are popped off the stack for coloring. Take the graphs in Figure 10 as an example. For each graph, every node initially has two neighbors. We first remove an arbitrary node, say J_1, then remove I_1, J_2 and finally I_2. In the coloring phase, we find the graph is 2-colorable. Such a heuristic method is approximate in the sense that it may declare a graph to be not n-colorable when it actually is. For loop tiling, such cases should be quite rare.

6.2.2 *Improving Colorability.* In case the heuristic method fails to prevent neighboring nodes from having the same color, we can improve colorability by *node splitting*. A nest of spatial loops can be split into two or more nests, as long as it does not break any T-independent data-dependence cycle. This is done by using a loop transformation known as *loop distribution*, or *loop fission* [Wolfe 1995].

Take the example in Figure 11(a). Its interference graph, shown in Figure 11(b), is not 2-colorable. However, if we distribute the spatial loops as shown in Figure 11(c), then the interference graph, shown in Figure 11(d), becomes 2-colorable. A detailed discussion of loop distribution is beyond the scope of this paper.

6.2.3 *Circular Loop Skewing.* Figure 12 shows an example of applying *circular loop skewing* [Wolfe 1995] to reduce backward dependence distances. The code skeleton in Figure 12(a) has a wrap-around stencil which is typical of PDEs with boundary conditions defined over a cylinder. Its iteration subspace is shown in Figure 12(b). Suppose that all J_i loops are free of loop-carried dependences and that the flow dependence from J_2 to J_1 has the distance vector $(1, 0)$. Within the same time step T, we make the following assumptions.

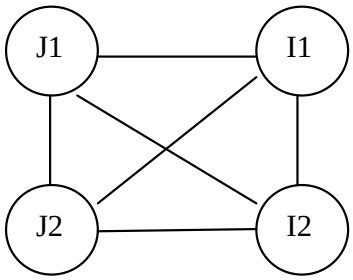
(1) The jth iteration of loop J_2 is flow-dependent on the $(j-1)$th, jth and $(j+1)$th iterations of J_1 for $2 \leq j \leq N-1$;

(2) the first iteration of loop J_2 is flow-dependent on the first, the second and the last iterations of J_1; and

(3) the last iteration of loop J_2 is flow-dependent on the $(N-1)$th, the Nth and the first iterations of J_1.

```
DO T = 1, ITMAX
  DO J1 = ...
  DO I1 = ...
    ... ← A(I1, J1)
    ... ← B(I1, J1)
  END DO
  END DO
  DO J2 = ...
  DO I2 = ...
    A(I2, J2) ← ...
    B(J2, I2) ← ...
  END DO
  END DO
END DO
```

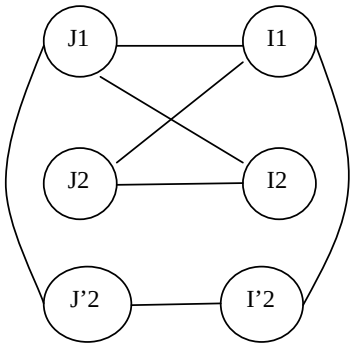
(a)



(b)

```
DO T = 1, ITMAX
  DO J1 = ...
  DO I1 = ...
    ... ← A(I1, J1)
    ... ← B(I1, J1)
  END DO
  END DO
  DO J2 = ...
  DO I2 = ...
    A(I2, J2) ← ...
  END DO
  END DO
  DO J2' = ...
  DO I2' = ...
    B(J2', I2') ← ...
  END DO
  END DO
END DO
```

(c)



(d)

Fig. 11. Loop distribution

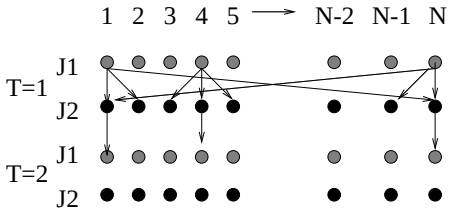
Following the algorithms in Section 4, we will get the minimum skew factor $S = N - 1$. However, we can reduce the backward dependence distances, and thus reduce the minimum skew factor, as follows. We change the iteration order of loop J_2 from $(1, 2, 3, \ldots, N)$ to the following. For $T = 1$, the order is $(2, 3, \ldots, N, 1)$. For $T = 2$, the order is $(3, \ldots, N, 1, 2)$. The rotation of the order continues with the increasing T-values. Similarly, we change the iteration order of loop J_1 from $(1, 2, 3, \ldots, N)$ to the following. For $T = 2$, the order is $(2, 3, \ldots, N, 1)$. For $T = 3$, the order is $(3, \ldots, N, 1, 2)$. The rotation of the order also continues with increasing T-values. Such a transformation will eliminate all of the dependences whose backward distances are $d = 1 - N$, resulting in a new skew factor $S = 1$. In a previous paper [Song and Li 1999], we presented an algorithm to formalize the idea illustrated above. We omit the details here.

6.3 Tiling with Speculative Execution

Iterative stencil loops may contain an IF statement which is executed at the end of each T-iteration. Such an IF statement often checks to see whether the iterative computation has converged. The T-loop may terminate before the iteration count has reached its maximum, $ITMAX$, if the convergence condition is already met.

```
DO T = 1, ITMAX
  DO J1 = 1, N
    < BODY_J1 >
  END DO
  DO J2 = 1, N
    < BODY_J2 >
  END DO
END DO
```

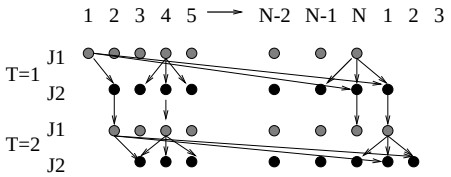
(a)



(b)

```
DO T = 1, ITMAX
  DO J1' = T, T + N - 1
    J1 = MOD(J1' - 1, N) + 1
    < BODY_J1 >
  END DO
  DO J2' = T + 1, T + N
    J2 = MOD(J2' - 1, N) + 1
    < BODY_J2 >
  END DO
END DO
```

(c)



(d)

Fig. 12. An example of circular loop skewing: (a) the original loop nest skeleton, (b) a part of the iteration-space graph before transformation, (c) the transformed loop nest skeleton, (d) a part of the iteration-space graph after transformation

The existence of such a convergence test in the T-loop defeats the loop tiling scheme which we have presented so far. This is because the control dependence imposed by the IF statement prevents the T-loop from being interchanged with inner loops. To overcome this difficulty, in this section we present an extension to our tiling method. The idea of the extension is as follows. We partition the $ITMAX$ iterations of the T-loop into a number of *T-chunks* such that the exit condition is tested only at the end of each T-chunk, instead of the end of each T-iteration. We then tile the individual T-chunks using the scheme presented in previous sections.

If the size of a T-chunk is smaller than $\Delta_{min} \equiv max\{\lceil \frac{B_k}{S_k} \rceil \mid 1 \leq k \leq n, S_k \neq 0\}$, no data elements will get fully reused within a tile traversal. Therefore, we let the minimum size of a T-chunk be Δ_{min}. The initial T-chunk size is set to $\Delta_0 = ITMAX/2$. If the exit condition tests false at the end of the initial T-chunk, the size of the next T-chunk is set to $\Delta_i = \frac{\Delta_{i-1}}{2} (i \geq 1)$. If the exit condition tests true, then the execution has overshot by a number of T-iterations. We then roll back the execution to the latest checkpoint and re-execute the loops with the T-chunk size further halved. As soon as the T-chunk size is equal to or smaller than Δ_{min}, the T-chunk is executed sequentially.

Since the T-chunk size is successively reduced by half until reaching Δ_{min} the maximum number of T-chunks to be executed is $O(log(ITMAX))$. The number of checkpoints inserted is therefore at most $O(log(ITMAX))$. The worst-case overhead

```
DO T = 1, ITMAX
  DO L_{1,1} = l_{1,1}, u_{1,1}
    ...
      DO L_{1,n} = l_{2,n}, u_{2,n}
        ...
      END DO /* L_{1,n} */
      DO L_{2,n} = l_{2,n}, u_{2,n}
        ...
      END DO /* L_{2,n} */
      ...
  END DO /* L_{1,1} */
  ...
  DO L_{m[1],1} = l_{m[1],1}, u_{m[1],1}
    ...
      DO L_{m[n]-1,n} = l_{m[n]-1,n}, u_{m[n]-1,n}
        ...
      END DO /* L_{m[n]-1,n} */
      DO L_{m[n],n} = l_{m[n],n}, u_{m[n],n}
        ...
      END DO /* L_{m[n],n} */
  END DO /* L_{m[1],1} */
    ...
 IF (exit condition = TRUE) THEN
  GOTO next
 END IF
END DO
next:
```

(a) An extended program model

```
ACCUM = 0
Δ = ITMAX/2
Initialize B to A.
DO CTRL = 1, ITMAX
  Execute the tiled T-chunk from T = (ACCUM + 1)
   to T = (ACCUM + Δ) .
  ECOND = exit condition for T = (ACCUM + Δ)
  IF (ECOND.EQ.TRUE) THEN
    Restore A from backup copy B.
    IF (Δ.LE.Δ_min THEN
      GOTO rollback
    ELSE
      Δ = Δ/2
      IF (Δ < Δ_min) Δ = Δ_min
    END IF
  ELSE
    ACCUM = ACCUM + Δ
    IF (ACCUM.EQ.ITMAX) GOTO next
    Copy A(or its odd copy C) to backup copy B.
    Δ = Δ/2
    IF (Δ < Δ_min) Δ = Δ_min
    IF ((ACCUM + Δ).GT.ITMAX) THEN
      Δ = ITMAX − ACCUM
    END IF
  END IF
END DO
GOTO next
roolback:
  Execute the original loop nest (Figure 13(a))
   from T = (ACCUM + 1).
next:
```

(b) The tiled code with speculative execution

Fig. 13. Tiling with speculative execution

of checkpointing, on the other hand, is $O(ITMAX)$ in terms of the number of time steps. The worst case takes place when the program turns out to converge in few time steps. Since in reality the number of executed iterations is generally near $ITMAX$, the rollback cost tends to be small. In Section 7 we shall see that the gain from the increased temporal reuse of the cache outweighs the loss due to checkpointing and over-shooting.

The transformed loops speculatively execute certain T-iterations. Thus, IF statements must be inserted to guard against potential exceptions such as overflow and divided-by-zero. If the possibility of exceptions is detected at run time, the execution rolls back to the latest checkpoint.

Formally, we extend the loop model defined previously in Figure 2(a) by including a loop exit test, as shown in Figure 13(a). The exit condition can only reference the variables defined either within the same T-iteration or outside the T-loop, and it should not be the source of any T-carried flow dependences. The exit condition should be monotonic. With monotonicity in the exit condition, we need to examine only the boundary of the chunk in order to decide whether rollback is necessary. We check for monotonicity using methods similar to those presented by Pugh [Pugh et al. 1996]. The following algorithm formalizes the idea discussed above.

Algorithm 6.1 Tiling with Speculative Execution
Input: A loop nest which conforms to the program model shown in Figure 13(a).
Output: A tiled loop nest.
Procedure:

—Transform the T-loop nest in Figure 13(a) into the code shown in Figure 13(b). Each T-chunk is tiled by applying the algorithms in Section 4. The arrays referenced in the exit condition are renamed in a way similar to array duplication.

□

In the transformed code, for every array A which is a source of T-carried flow dependences, we create a *backup copy*, B, which initially has the same values as A. The code uses a variable $ACCUM$ to accumulate the total T-iterations executed so far. In every iteration of the outermost `DO` loop, after executing Δ steps of the tiled program, the rollback condition is checked. If the execution must rollback, the values stored in B are restored and the computation resumes from the beginning of the T-iteration for $T = (ACCUM + 1)$, which is the latest checkpoint. Otherwise, A is copied to the backup copy B. If the execution reaches the last T-iteration, the outermost `DO` loop terminates.

7. EXPERIMENTAL EVALUATION

We have implemented the tiling techniques presented above in the Panorama compiler [Gu et al. 1997]. In our experiments so far, we have applied the technique to the following test programs which contain iterative stencil loops:

—Jacobi, which is a small kernel often used as an example of relaxation methods.

—Loop No. 18 (LL18) from the famous set of Livermore Loops.

—Two signal processing kernels, called `acoustic2d` and `acoustic3d` respectively, from the BLITZ++ benchmark suite [Object-Oriented Scientific Computing]. The `acoustic2d` program contains two levels of spatial loops, and `acoustic3d` contains three levels.

—Ten programs from a PDE solver package called MUDPACK from the University Corporation for Atmospheric Research [Admas]. Five of them contain two levels of spatial loops and the rest contain three levels.

—A program, `tomcatv`, from the industrial SPEC95 benchmarks.

—A program, `swim`, which has two versions, namely `swim95` from SPEC95 benchmarks and `swim2000` from SPEC2000 benchmarks. The main difference between these two versions is in the data size. In `swim2000`, each array contains 1335×1335 double-precision floating point numbers. In `swim95`, each array contains 513×513 single-precision floating point numbers.

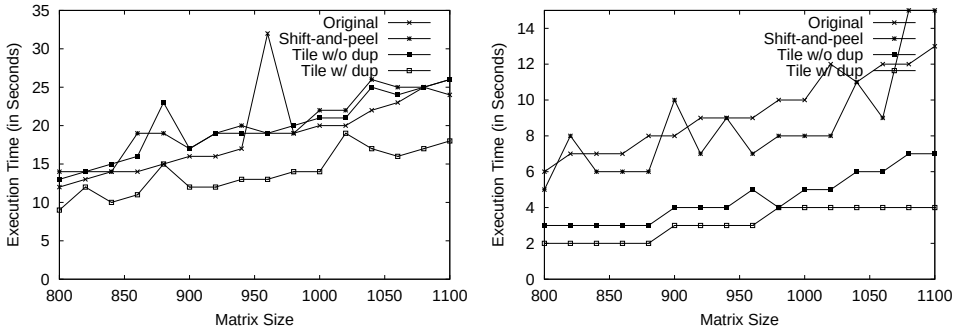
We run the test programs on a SUN Ultra I uniprocessor workstation and on a single MIPS R10K processor of an SGI Origin 2000 multiprocessor machine. We compared the codes tiled by our scheme with the original codes optimized by the native compilers. To highlight the benefit of temporal locality achieved across different time steps by our method, we also compare with a method called *shift-and-peel* [Manjikian and Abdelrahman 1997] which combines loop peeling, loop fusion and cache partitioning to aggressively exploit data locality at inner loop levels.

On the R10K, all codes are compiled using the native MIPSpro compiler to generate the machine code, with the highest optimization level "-O3". The MIPSpro compiler has certain loop tiling capability at the "-O3" level, but it is unable to deal with iterative stencil loops as our scheme does. When the codes tiled by shift-and-peel and by our scheme are compiled by the native compiler, we switch off the native compiler's own loop-tiling option. On the Ultra I, all codes are compiled using the native compiler, with the "-xO4" optimization switch turned on. This is

Table I. Hardware Parameters

	Ultra I			R10K		
	Primary cache	secondary cache	TLB	Primary cache	secondary cache	TLB
Size	2K	64K	64	4K	512K	64
Line size	2	8	1K	4	16	4K
Associativity	1	1		2	2	
Miss penalty	6	45		9	68	

Notes: The TLB size is in the number of entries. All other sizes are in the number of data elements. Each data element takes eight bytes. Miss penalties are in the number of CPU cycles.



(a) On the Ultra I

(b) On the R10K

Fig. 14. Execution time (in seconds) of Jacobi on the Ultra I and the R10K with various matrix sizes

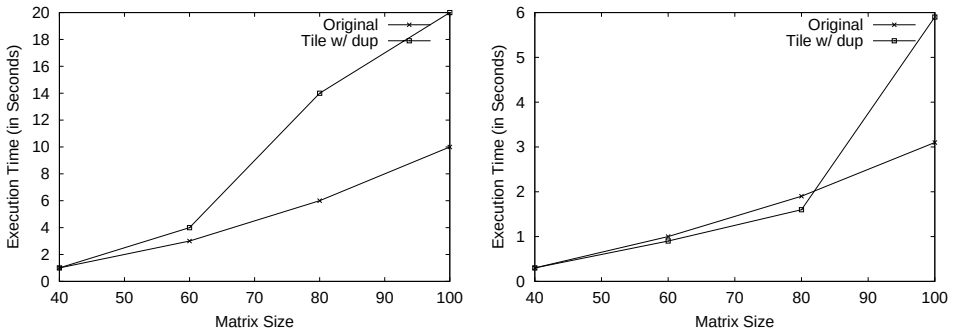
the highest optimization level without feedback from profiling. The SUN compiler on our machine does not perform loop tiling at any optimization level.

Table I lists the hardware parameters on the Ultra I and the R10K. On both machines, the secondary cache stores both instructions and data. We use the hardware counters on the R10K and the Ultra I to measure the cache miss rate.

We applied the speculation technique (in Section 6.3) to Jacobi and `tomcatv`. (The rest of the test programs do not contain exit conditions in the time-step loop.) Circular loop skewing does not have any effect except for `swim`. In the rest of this section, we provide further experimentation details for the individual test programs.

7.1 Execution Time Improvement

The Jacobi Kernel. We arbitrarily fix ITMAX to 100 and vary the input matrix size arbitrarily from 800 to 1100, in increments of 20. The skew factors are $\vec{S} = (1, 1)$ for tiling with array duplication and $\vec{S} = (2, 2)$ for tiling without array duplication. For all test cases, Jacobi always runs to the maximum time step, i.e., 100. Figures 14(a) and (b) show the performance result for different versions of Jacobi code on the Ultra I and the R10K respectively. Our scheme chooses to tile with array duplication for all the test cases on both machines. (In all figures and tables in this subsection, the label "Original" stands for the original codes, "Shift-and-peel" for the shift-and-peel codes, "Tile w/ Dup" for our codes tiled



(a) On the Ultra I

(b) On the R10K

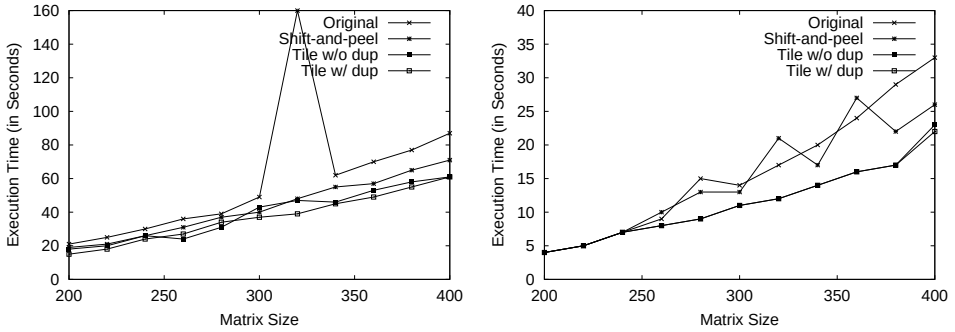
Fig. 15. Execution time (in seconds) of Jacobi on the Ultra I and the R10K with small matrix sizes

with duplication and "Tile w/o Dup" for our codes tiled without duplication.)

Figure 14 shows that, when compared with the other three schemes, tiling with array duplication performs equally well or better in all cases on both machines. On the Ultra I, our tiled codes achieve a speedup of 1.00 to 2.46 over the original codes. Our tiled codes achieve a speedup of 1.16 to 1.73 over shift-and-peel codes. On the R10K, our tiled codes achieve a speedup of 1.75 to 4.00 over the original codes and achieve a speedup of 1.50 to 4.00 over shift-and-peel codes.

Loop tiling incurs a certain execution overhead. The most prominent potential overhead may be the potential degradation of software pipelining. The relatively complex loop construct after loop tiling may reduce the effectiveness of the back-end compiler to generate efficient software pipelining code for the innermost loops. For large data sizes, i.e. for higher loop iteration counts, the benefit of increased data locality in the tiled loops tends to overwhelm the penalty. However, for small data sizes which fit in the secondary cache, the penalty for primary-cache misses is significantly lowered. Hence, the benefit of increased locality may be insufficient to offset the penalty. Such a possibility is highlighted by the results shown in Figure 15 which are collected by choosing several array sizes which allow the total data size to fit in the secondary cache. (To make the test program run long enough time, we increased *ITMAX* to 10000.) In this figure, we observe that the tiling overhead indeed causes degraded performance on the Ultra I, although it does less so on the R10K. The array-size cutoff for the tiling decision seems to depend on how the native compiler performs scalar and instruction-level parallelism optimizations. In practice, one can find these cutoff numbers by experimenting with sample loops on the target machine. Two versions of code can then be produced, one with tiling (for large data sets) and one without tiling (for small data sets).

The LL18 Kernel. We fix ITMAX to 400 and choose the input matrix sizes from 200 to 400, in increments of 20. We choose smaller matrix sizes than those in Jacobi because LL18 has nine arrays instead of two. Using the same array sizes as in Jacobi would result in extremely small tile sizes. Our scheme duplicates two arrays, *ZR* and *ZZ*, and gets $\vec{S} = (1, 1)$. Array duplication increases the memory



(a) On the Ultra I (b) On the R10K

Fig. 16. Execution time (in seconds) of LL18 on the Ultra I and on the R10K with various matrix sizes

requirement by 18%. Without array duplication, we would have $\vec{S} = (2, 2)$.

Figure 16 shows the performance result for various versions of LL18 on the Ultra I and the R10K respectively. On the Ultra I, our tiled codes achieve a speedup of 1.25 to 4.10 over the original codes and a speedup of 1.08 to 1.27 over the shift-and-peel codes. On the R10K, our tiled codes achieve a speedup of 1.00 to 1.71 over the original codes and a speedup of 1.00 to 1.75 over the peel-and-fusion codes. We observe that, in most cases, the execution speed of codes tiled with and without array duplication is very similar. Array duplication enjoys a slight advantage on the Ultra I.

Array duplication may increase the program data size. If such an increase causes the data to exceed the memory capacity, then the arrays should not be duplicated. For LL18, we did an experiment with $N = 1250$. The original program can be executed in-core on our Ultra I machine. However, the tiled program with array duplication has to be executed out-of-core and, as a result, it does not finish in a reasonable time. Of course, one can apply multi-level tiling to the loops, so that the tile size at the higher level is chosen to fit the page size. Within the page size, one can then tile for the caches. Even so, however, the code with array duplication is still expected to generate more page faults than the code without array duplication. Therefore, duplication should be avoided when it causes the program to execute out-of-core. We do not perform further experiments on this effect for this paper.

Table II. Execution time (in seconds) of `tomactv`

Test Program	R10K		Ultra I	
	Exec. Time	Speedup	Exec. Time	Speedup
Original	155.1	1.00	405	1.00
Shift-and-peel	136	1.14	356	1.14
Tile w/ Dup	73.8	2.10	310	1.31
Tile w/o Dup	73.9	2.10	328	1.23

Table III. Execution time of `tomcatv` on the R10K with various convergence tests

Org. Iter.	Org. Time	Tiled Iter.	Tiled Time	Speedup
11	7.3	717	71.2	0.10
38	12.7	744	79.7	0.16
105	23.6	717	74.9	0.32
359	76.6	787	74.9	1.02
513	109.3	751	69.8	1.52
620	123.6	764	77.2	1.60
731	157.1	773	77.5	2.03
750	155.1	750	73.8	2.10

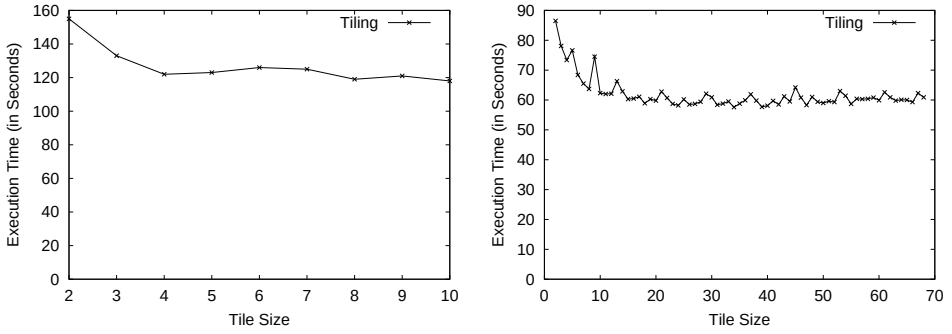
The tomcatv Program. The program `tomcatv` has seven N-by-N arrays in its iterative stencil loops. Among these, two arrays, X and Y, are duplicated when we tile the loops with array duplication. The duplication increases the memory usage by 29%. Loop interchange and array transpose are applied before tiling to remove incompatibility between several loops at the same level. Due to long backward dependence distances at the higher levels of spatial loops, tiling can be applied profitably only to the highest level of spatial loops. The skew factor is $S_1 = 1$ with array duplication and $S_1 = 2$ without duplication. On the R10K, the tile size B_1 is computed as 51 with array duplication and as 66 without duplication. On the Ultra I, the tile size B_1 equals 11 and 14, respectively, for loops tiled with and without duplication. On both machines, our compiler chooses tiling with duplication. Using the reference input data, `tomcatv` always runs to the maximum time step, i.e., 750. The tiled code with speculative execution executes five checkpoints on the R10K both with and without array duplication. It executes seven checkpoints on the Ultra I. The performance data is shown in Table II.

We have also studied the effect of rolling back T-iterations in case of misspeculation by changing the convergence test condition in `tomcatv`. Table III shows the results. The "Org. Iter." column lists the number of iterations executed in the original code before the computation converges according to the new condition. The "Org. Time" column lists the execution time corresponding to the "Org. Iter." column. The "Tiled Iter." column lists the number of iterations executed in total by the tiled code before the computation converges. The "Tiled Time" is the execution time corresponding to the "Tiled Iter." column. The "Speedup" column lists the speedup of the tiled code over the original codes. It can be seen that if the convergence happens in the very beginning, the tiled code is slower than the original code. Otherwise, the tiled code is faster.

The swim Program. For `swim`, no arrays need to be duplicated. Long backward dependence distances exist at two levels of the spatial loops. We reduce the distances at the highest level using circular loop skewing, but we cannot do so at the lower level. Hence, only the highest level of the spatial loops are tiled. The skew factors are computed as $S_1 = 2$. For the `swim95` version, the tile sizes are computed as $B_1 = 34$ on the R10K and $B_1 = 5$ on the Ultra I. For `swim2000`, we get $B_1 = 25$ on the R10K and $B_1 = 3$ on the Ultra I. (The tile sizes differ because of the different iteration counts of the innermost spatial loops which are not tiled.) Table IV shows the performance results, where the row labeled by "our tiled" shows the results of the program transformed by our tiling techniques. Shift-and-peel does not apply

Table IV. Execution time (in seconds) of `swim`

Test Program	R10K		Ultra I	
	Exec. Time	Speedup	Exec. Time	Speedup
`swim95` (original)	92	1.00	241	1.00
`swim95` (our tiled)	58	1.59	127	1.90
`swim2000` (original)	619.5	1.00	2286	1.00
`swim2000` (our tiled)	380	1.63	1700	1.34



(a) On the Ultra I (b) On the R10K

Fig. 17. Execution time (in seconds) of `swim95` on the Ultra I and the R10K with different tile sizes

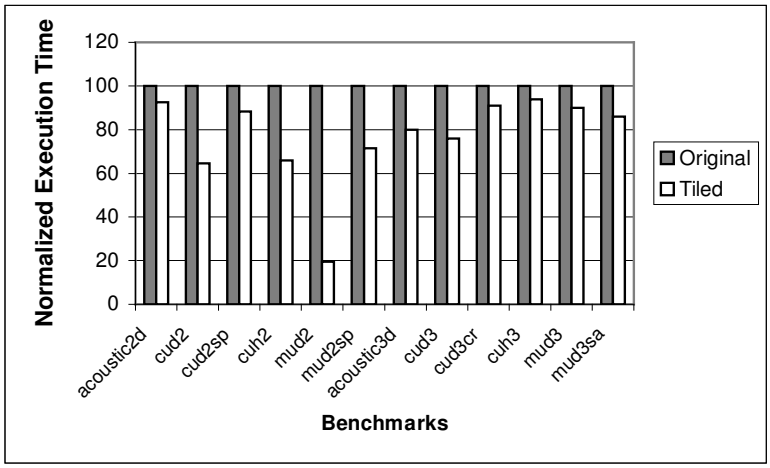
directly to `swim` due to the long backward dependence distances.

To assess how good our tile-size selection algorithm is, we run `swim95` with different tile sizes ranging from 2 to twice of what our algorithm chooses. Figure 17 shows the results. The performance is not very sensitive to the tile size and our chosen tile sizes are very close to the best ones shown by the experiments.

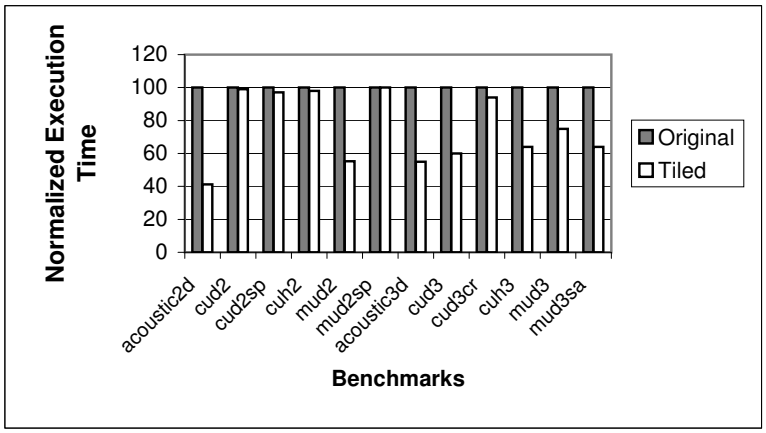
Acoustic and MUDPACK programs. For `acoustic2d` and `acoustic3d`, we use the reference input from the benchmark provider. MUDPACK contains a set of subroutines for elliptic partial differential equations [Admas]. It also contains test drivers which we can use to measure the performance. Combining the test drivers with various subroutines, we obtain ten sample test programs. Five of them, namely `cud2`, `cud2sp`, `cuh2`, `mud2` and `mud2sp`, contain two levels of spatial loops. The rest, `cud3`, `cut3cr`, `cuh3`, `mud3`, and `mud3sa`, contain three levels of spatial loops. We increase the test data size for these programs so that the execution time will last at least several seconds. Procedure cloning and loop embedding [Wolfe 1995] are applied before the iterative stencil loops are tiled. Shift-and-peel is not applied to these six programs because its cache-partitioning step cannot be applied in order to compute the tile size. Figure 18 shows the performance result, where the tiled code achieves a speedup between 1.00 and 5.06.

7.2 Memory Reference Characteristics

On the R10K, in addition to the execution time, we have also collected data that show the memory reference characteristics of different versions of programs. These



(a) On the Ultra I



(b) On the R10K

Fig. 18. Normalized execution time (in seconds) of several new PDE solvers on the Ultra I and the R10K

Table V. Cache misses in Jacobi, LL18, `acoustic2d`, `mud2`, `tomcatv` and `swim95` (LS, PM and SM are counted in millions)

Test Program	Original				Shift-and-peel			
	LS	PM	SM	PMR	LS	PM	SM	PMR
Jacobi (N = 1000)	553.5	106.3	24.7	0.19	638.8	100.1	12.2	0.16
LL18 (N = 300)	1205.5	201.4	15.7	0.17	1182.6	216.5	13.2	0.18
`acoustic2d`	2034.3	177.9	28.1	0.09	-	-	-	-
`mud2`	4197.8	721.9	75.5	0.17	-	-	-	-
`tomcatv`	10826	1523	177	0.14	10033	1312	138	0.13
`swim95`	11706	871	166	0.07	-	-	-	-
Test Program	Duplication				Non-duplication			
	LS	PM	SM	PMR	LS	PM	SM	PMR
Jacobi (N = 1000)	712.4	8.9	1.2	0.012	806.0	13.9	1.4	0.017
LL18 (N = 300)	1234.0	257.0	0.8	0.21	1199.9	162.4	0.4	0.14
`acoustic2d`	-	-	-	-	2285.7	172.2	1.3	0.08
`mud2`	-	-	-	-	4232.6	757.7	47.5	0.18
`tomcatv`	9210	1150	3.5	0.12	9032	1067	3.6	0.12
`swim95`	-	-	-	-	9305	819	5.8	0.088

data are collected using the *perfex* library based on R10K performance counters. We show this data for Jacobi, LL18, tomcatv, swim95, one Acoustic program, namely `acoustic2d`, and one program from MUDPACK, namely `mud2`. In this table, the 'LS' column lists the number of load-store instructions executed at run time. The 'PM' column lists the number of primary cache misses. The 'SM' column lists the number of secondary cache misses, and the 'PMR' column lists the primary cache miss ratio. The data for Jacobi and LL18 are collected with the array size N arbitrarily chosen to be 1000 and 300 respectively. For Jacobi, we see that the primary cache miss rate is reduced by an order of magnitude. For all six test programs, the number of secondary cache misses is dramatically reduced by our tiling scheme. The gain in temporal locality across T iterations as the result of array duplication clearly outweights the penalty due to the increased memory requirement.

7.3 Effect of Array Padding

The array padding scheme described in Section 5 is applied when we perform tiling on the test programs. For inter-array padding, we place the arrays which need this padding in the same Fortran common block and then insert an artificial array, whose size is determined by the padding scheme, between each pair of arrays to be padded. In order to examine the effect of array padding, we run the programs listed in Table V on the Ultra I with and without array padding. The results are shown in Table VI. The label "Original" means the original programs before tiling. The label "w/o pad" means the tiled programs before array padding. Lastly, the label "w/ pad" means the tiled programs after array padding. We note that the degree of improvement by padding depends on how poorly the original array sizes match the cache. None of the test cases we listed here are extremely pathological, yet, the improvement by array padding is still noticeable.

7.4 Discussion

It is also possible to tile the time-step loop. Tiling in this way can potentially increase the data reuse between different tile traversals. However, this also increases the total number of tile traversals. Since the memory references in the initial tile

Table VI. Effect of Array Padding

Test Program	Execution Time (Seconds)			Speedup by Padding
	Original	w/o pad	w/ pad	
Jacobi (N = 1000)	20	15	14	7.1%
LL18 (N = 300)	49	40	37	8.1%
`acoustic2d`	129	122	118	3.4%
`mud2`	667	134	130	3.0%
`tomcatv`	405	344	310	11%
`swim95`	241	152	127	20%
`swim2000`	2286	1761	1700	4%

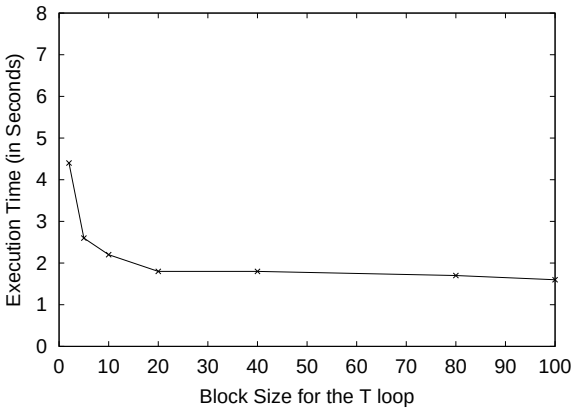


Fig. 19. Execution time of Jacobi on the R10K with different T-loop block sizes

in each tile traversal likely incur cache misses, creating more tile traversals tends to result in a net increase in cache misses. The execution time when both the spatial loops and the time-step T-loop are tiled with different T-loop block sizes, where $N = 1000$ and $ITMAX = 100$, are shown for the Jacobi example (with the convergence test removed) in Figure 19. The block size 100 amounts to no blocking for the T-loop. One can see that blocking the T-loop is not beneficial in the case of Jacobi. We do not perform further experiments with the idea of tiling the T-loop for this paper.

In this paper, loop fusion and forward substitution are applied to `Jacobi` to eliminate the temporary array *temp*, which reduces the memory overhead introduced by array duplication. A more general technique of array contraction [Song et al. 2001] may be applied to a broader class of programs to reduce the sizes of temporary arrays. Array contraction reduces array sizes from a higher dimension to a lower dimension, sometimes even to scalars. For example, several local arrays in `tomcatv` can be contracted from two dimensions to one dimension after tiling. We do not explore such opportunities for this paper, as it requires an additional mathematical framework and a number of additional compiler schemes [Song et al. 2001].

8. RELATED WORK

Loop tiling has been used in two important contexts for high performance computing. On multiprocessors, it can be used to reduce inter-processor communication

[Xue 2000; Boulet et al. 1999]. For machines which employ cache memories, it can be used to improve the cache hit ratio both in sequential programs [Wolfe 1995] and in parallel programs [Sarkar 1998]. Our paper is aimed at cache performance improvement for sequential programs. We use loop tiling to reorder loop iterations and partition them into *tiles* such that the data accessed in each tile can fit in the cache and be reused multiple times.

Previous publications that are closely related to this paper can be divided into two categories. One category concerns loop transformations and data transformations. The other concerns the removal of set conflicts in the tiled loops. We discuss closely-related work in these two categories separately.

8.1 Loop Transformations

There exists an interesting relationship between loop tiling and loop fusion. Loop fusion is commonly performed to exploit data reuses between adjacent loops which appear at the same loop level [Manjikian and Abdelrahman 1997; Kennedy and McKinley ; Kennedy 2000; Ding and Kennedy 2001]. When performed this way, unlike the tile scheme presented in this paper, loop fusion does not exploit temporal data locality across different time steps in iterative stencil loops.

On the other hand, it is possible to fuse entire nested loops into a perfect nest. After that, one may be able to use a previously proposed loop skewing method [Wolf 1992] to make the fused loop nest fully permutable at certain outer levels. Loop tiling can then be applied to those outer loop levels. Hence, as we mentioned in Section 2.1, tiling can be performed with or without first fusing the spatial loops into a single perfect nest. Recently proposed methods such as *loop embedding* [Ahmed et al. 2000] and *iteration space slicing* [Pugh and Rosser 1999] use various heuristics to find legal fusions which can improve data reuse. We have shown in this paper that, for iterative stencil loops, the spatial loops generally must be skewed over the time steps before they can be tiled. Previous publications have separated the issue of loop fusion from loop skewing and loop tiling. Thus, they do not analyze the effect of loop alignment (during loop fusion) on the later step of skewed tiling. In this paper, however, we have shown that loop alignment and loop skewing must be considered simultaneously in order to maximize the benefit of tiling. We have presented a polynomial-time algorithm to tile iterative stencil loops directly with minimum skew factors. We also have also presented a number of techniques to reduce the skew factors. Among those, the array duplication technique can be optimally performed, based on the same algorithm which finds the minimum skew factors.

For a special class of memory-constrained loops (which are not iterative stencil loops), Cociorva et al. have shown that there exists a tradeoff between loop fusion and loop tiling [Cociorva et al. 2001]. This class of loops originates from multidimensional integration in computational chemistry. On one hand, loop fusion can reduce the total number of arithmetic operations in these loops. On the other hand, the kind of loop fusion performed may disable loop tiling and thus may reduce temporal data locality. They present a method to make the tradeoff.

There exist numerous publications on the tiling of a class of loops often known as *linear algebra* loops, which include familiar examples such as matrix multiplication and Gausian elimination. This class of loops has data-dependence characteristics

that are distinctly different from iterative stencil loops. The most prominent difference is that they normally do not require loop skewing. To an extent, our array padding analysis can also be applied to this class of loops. However, the rest of the techniques discussed in this paper mainly target iterative stencil loops only.

After the publication of our initial work on tiling iterative stencil loops [Song and Li 1999], several authors have published new contributions to similar problems. Wonnacott develops a scheme called *time skewing* which adopts a value-based flow analysis to optimize for memory locality [Wonnacott 2002]. His method first performs full array expansion and forward substitution, and then it recompresses the expanded array while preserving data dependences. His method focuses on 1-D tiles. Jin et al. have manually applied recursive blocking, in conjunction with skewing, to two numerical kernels (SOR and Jacobi) and a supercomputing application (*SMG98*) [Jin et al. 2001]. Recursive blocking uses recursive calls to produce recursively-embedded multi-level tiles. It is still unclear how this method affects conflict misses in iterative stencil loops. If this problem is solved, one can use the recursive blocking method in our automatic tiling framework.

There exist techniques which use program transformations other than loop tiling to improve data locality in imperfectly nested loops. Kodukula and Pingali propose a matrix-based framework to represent transformations of imperfectly-nested loops [Kodukula and Pingali 1996] including permutation, reversal, skewing, scaling, alignment, distribution and jamming. Such techniques seem to work quite well on loops which are not iterative stencil loops, although they do not seem to exploit temporal locality in iterative stencil loops. Kodukula et al. have also proposed a data-centric technique, called *data shackling* [Kodukula et al. 1997], which blocks the arrays based on data flow analysis and then forms new loop nests to compute the array values block-by-block. Although it can handle certain imperfectly-nested loops, their method does not apply to the stencil computations handled in our work, because updating one block will destroy the boundary data necessary for its adjacent blocks. Pugh and Rosser present a technique called *iteration space slicing* to find the set of computations which affect the value of a given array element [Pugh and Rosser 1999; Rosser 1998]. By reordering the computation based on such slicing cache locality may be improved. This technique can expose many opportunities to improve data locality. However, how to optimally choose among all possible opportunities remains a challenging issue. Collard proposes a method to speculatively execute while-loops on parallel machines [Collard 1994]. His objective is to increase parallel processor utilization while ours is to tile loops for better memory performance on uniprocessors. We use quite different algorithms. Pugh et al. present a method to handle exit conditions for iterative application in parallel environment [Pugh et al. 1996]. Our method to verify monotonicity of exit conditions is similar to that of Pugh et al. Both works by Collard and by Pugh et al. present additional interesting ideas to handle loop exit conditions, which may be incorporated in our future work.

A number of authors have worked on memory-cost analysis for loop transformations, including tiling. Such analysis can be quite useful in many cases, although the techniques proposed so far have not dealt with imperfectly-nested loops such as those covered in this paper. Among these, Strout et al. discuss the minimum storage requirement to allow flexible loop scheduling such as tiling [Strout et al.

1998]. Ferrante *et al.* provide closed-form formulas that bound the number of array accesses and the number of cache lines within a loop nest, thus providing an estimate of the number of cache misses in a loop nest [Ferrante et al. 1991]. Temam *et al.* present an algorithm to estimate the number of cache misses, especially interference misses [Temam et al. 1994]. Ghosh *et al.* present cache miss equations (CME) to count the number of cache misses in order to guide optimization [Ghosh et al. 1998]. Mitchell et al. use matrix multiplication as an example to show that both the TLB misses and the cache misses must be considered simultaneously in order to achieve the best performance [Mitchell et al. 1998]. However, they provide no formal algorithms to select tile sizes.

8.2 Interference-Removal Techniques

For the removal of interferences between variable references in the cache, the main differences between the approach presented in this paper and the previous approaches are two-fold. First, our approach removes self interferences for general m-dimensional array tiles, where $m > 1$, while previous methods mostly target one or two-dimensional arrays. Second, our approach is aimed at minimizing capacity misses and conflict misses simultaneously, while previous methods generally cannot do so. In the rest of this section, we give additional detailed accounts of related methods. Minor differences between our approach and the previous methods are pointed out along the way.

LRW. The LRW tile-selection scheme by Lam, Rothberg and Wolf [Lam et al. 1991] has led to a number of follow-up schemes in recent years. LRW considers two-dimensional array tiles and it uses an algorithm to compute the largest square tile without *self interferences*, i.e., set conflicts that are confined within a single tile. The algorithm sequentially increases the size of the square tile, checking to see whether different array elements will conflict at the same cache line. The algorithm runs in $O(N/\sqrt{C})$ time. The experimental results in the LRW paper show that the cache is often under-utilized when the conflict-free tile is required to be square. Therefore, the theoretical minimum cache misses often cannot be achieved. The paper states that their algorithm can be easily extended to find the largest rectangular block. Indeed, several follow-up schemes try to extend LRW by allowing rectangular tiles.

TSS. The TSS scheme by Coleman and McKinley [Coleman and McKinley 1995] is the first to use Euclidean remainders to study the mapping of rectangular tiles to the cache. The TSS scheme selects a number of candidate tile sizes which guarantee the absence of self interferences. The scheme then uses a probabilistic model to estimate cross interferences. It chooses the tile size for which the estimated *conflict misses* are minimum.

The TSS scheme is limited to two-dimensional tiles and it is possible for TSS to choose a narrow tile which incurs minimal conflict misses at the expense of heavy capacity misses. Another difference, although a minor one, between TSS and our approach is that TSS requires the declared length of the array column to be no greater than the cache size. This is because TSS begins by initializing the tile-column size to the array-column length. In practice, it is quite possible for the array-column length to exceed the cache size. For example, the primary cache size

is 1024 cache blocks on both Sun Microsystem's Ultra I microprocessor and SGI's R10K microprocessor. To take the advantage of the spatial locality, the column size of the array tile is preferred to be a multiple of the cache-block size. Thus, in order for TSS to work, the array-column length cannot be greater than 1024 divided by the number of arrays referenced in the given loop nest.

TLI. In order to improve TSS, one can extend it by enumerating all array-tile sizes allowed by the cache size. The TLI scheme is such an extension to TSS [Chame and Moon 1999]. For each tentative tile size, TLI sequentially "stamps" the tiles on the cache in order to find out whether self interferences occur. This can be viewed as a way to simulate how the array data will be mapped to cache locations. Suppose the extent of each array dimension is no less than the cache size. The cost of such an extended scheme can be estimated as follows. For one-dimensional array tiles, the tile size is simply chosen such that its total footprint covers the cache as much as possible. This can be done in constant time. For two-dimensional array tiles, the complexity of the extended scheme is $O(C \cdot H_c)$, where C is the cache size in the number of array elements and $H_c \equiv 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{C}$ is a harmonic number. Recall that $lim_{C\to\infty}(H_c - lnC) = \alpha$, where α is the Euler's constant. Hence the complexity can also be written as $O(C \cdot ln(C))$. For m-dimensional array tiles, $m > 2$, the complexity is $O(C^2 ln^{m-2}(C))$. The resulting tile still does not minimize the capacity misses and the conflict misses simultaneously.

Array Padding. The methods mentioned above adjust the tile size in order to remove conflict misses. This is at the expense of increased capacity misses. Bacon *et al.* [Bacon et al. 1994] propose array padding as a new approach to removing conflict misses. Within the scope of the innermost loop in a given loop nest, they study how to use both intra-array padding (called *dimension padding* in their work) and inter-array padding to remove conflicts between two static references. These can be references to the same array or to different arrays. Although their method does not directly target a tiled loop nest, it shows the general idea of how the array declarations can be changed in order to alter the mapping between the array data and the cache sets.

Panda *et al.* are the first to use array padding to remove self interferences in array tiles [Panda et al. 1999]. Their method, called DAT, first tries to find a tile size to maximize the cache capacity and then tries intra-variable and inter-variable padding to eliminate self interference and minimize cross interference conflicts. The DAT method, however, exhaustively enumerates pad sizes and essentially simulates the cache to see whether self interferences occur under a given tile size and a pad size. The compile-time cost of such an exhaustive approach is proportional to C^m for m-dimensional array tiles, where C is the cache size measured by the number of array elements.

3D Padding by Rivera and Tseng. All previous methods mentioned above remove self interferences for two-dimensional array tiles only, although array tiles of higher dimensions have seen increased use in numerical computing due to the increased processor speed and cache size. Rivera and Tseng have conducted a number of experiments on array padding for 3-D array tiles [Rivera and Tseng 2000]. (There is a statement in their paper claiming that a 3-D array tile is free of self interferences

if GCD$(N_i, C) = D_i$ in each tile dimension i. This statement should be revised according to our Lemma 5.2.)

Blocked Array Layouts. A number of authors have investigated the impact of nonlinear array layouts on hierarchical memory systems [Chatterjee et al. 1999; Chatterjee et al. 1999; Wise et al. 2001; Park et al. 2002; Andersen et al. 1999]. Such nonlinear array layouts go beyond the conventional column-major and row-major linear array layouts. Nonlinear layouts are very attractive as a way to eliminate self interferences, because they allocate the elements in the same array tile consecutively in the memory. On the other hand, nonlinear layouts can make array indexing substantially more complex. In addition to the increased overhead in memory-address computation, complex indexing can potentially degrade the effectiveness of software pipelining performed by the back-end compiler. This interesting technique needs to be studied carefully in our future work.

9. CONCLUSION AND FUTURE WORK

In this paper, we have presented a scheme to tile a class of iterative stencil loops. Central to this scheme is a systematic way to skew tiles and tile traversals so that data dependences are satisfied despite the fact that the given loops may be imperfectly nested. We present a graph-theoretical algorithm, which takes polynomial time, to determine the minimum skew factor at each loop level. A number of techniques are also presented to reduce the minimum skew factors in the given loop nest. We use a memory-cost analysis to derive the optimal tile size. Given the tile size, an efficient multidimensional *array-padding* scheme is applied to eliminate reference interferences. Experiments performed on sixteen test programs show that the overall tiling scheme can improve the program execution speed quite significantly.

We should emphasize that the work presented in this paper mainly targets iterative loops which implement relaxation methods in numerical computing. Other recent publications have discussed loop tiling for different kinds of imperfectly-nested loops. It remains an intriguing challenge to develop a unified scheme to tile arbitrary loops which are imperfectly nested.

ACKNOWLEDGEMENTS

This work is sponsored in part by the National Science Foundation through grants CCR-950254, CCR-9975309, MIP-9610379 and by the Purdue Research Foundation. The authors are grateful to Greg Frederickson for pointing out the reference to the minimum cost-to-time ratio problem, to Sam Midkiff for proofreading, and to the anonymous reviewers for valuable comments which help improve the paper's quality. Rong Xu helped implement the value-based flow dependence distance computation. Ruihao Zhang implemented part of the tiled-code generation. Parts of the material in this paper were presented at the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation, Atlanta, Georgia, May 1999 and at the 17th IEEE International Parallel and Distributed Processing Symposium, Nice, France, April, 2003.

REFERENCES

Admas, J. C. *MUDPACK: Multigrid Software for Elliptic Partial Differential Equations*. http://www.scd.ucar.edu/css/software/mudpack/.

Ahmed, N., Mateev, N., and Pingali, K. 2000. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. In *Proceedings of the 2000 International Conference on Supercomputing*. Santa FE, NM, 141–152.

Ahuja, R., Magnanti, T., and Orlin, J. 1993. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall Inc., Englewood Cliffs, New Jersey.

Allan, V., Jones, R., Lee, R., and Allan, S. 1995. Software pipelining. *ACM Computing Surveys 27,* 3 (September), 367–432.

Allen, J. R. and Kennedy, K. 1984. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems 9,* 4 (October), 491–542.

Andersen, B. S., Gustavson, F. G., Wasniewski, J., and Yalamov, P. Y. 1999. Recursive formulation of some dense linear algebra algorithms. In *SIAM Conference on Parallel Processing for Scientific Computing*. San Antonio, Texas.

Anderson, J. M., Amarasinghe, S. P., and Lam, M. S. 1995. Data and computation transformation for multiprocessors. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Santa Barbara, CA, 166–178.

Bacon, D., Chow, J.-H., Ju, D., Muthukumar, K., and Sarkar, V. 1994. A compiler framework for restructuring data declarations to enhance cache and tlb effectiveness. In *Proceedings of CASCON'94*. Toronto, Ontario.

Blume, W. and Eigenmann, R. 1998. Non-linear and symbolic data dependence testing. *IEEE Transactions of Parallel and Distributed Systems 9,* 12 (December), 1180–1194.

Boulet, P., Dongarra, J., Robert, Y., and Vivien, F. 1999. Static tiling for heterogeneous computing platforms. *Parallel Computing 25*, 547–568.

Briggs, P., Cooper, K., Kennedy, K., and Torcson, L. 1989. Coloring heuristics for register allocation. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*. 275–384.

Burger, D. C., Goodman, J. R., and kägi, A. 1996. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23rd International Symposium on Computer Architecture*. Philadelphia, PA, 78–89.

Chame, J. and Moon, S. 1999. A tile selection algorithm for data locality and cache interference. In *Proceedings of the Thirteenth ACM International Conference on Supercomputing*. Rhodes, Greece, 492–499.

Chatterjee, S., Jain, V., Lebeck, A., Mundhra, S., and Thottethodi, M. 1999. Nonlinear array layouts for hierarchical memory systems. In *Proceedings of International Conference on Supercomputing*. Rhodes, Greece, 444–453.

Chatterjee, S., Lebeck, A., , Patnala, P. K., and Thottethodi, M. 1999. Recursive array layouts and fast parallel matrix multiplication. In *Proceedings of 11th ACM Symposium on Parallel Algorithms and Architectures*. Saint Malo, France.

Cociorva, D., Wilkins, J. W., Lam, C., Baumgartner, G., Ramanujam, J., and Sadayappan, P. 2001. Loop optimization for a class of memory-constrained computations. In *Proceedings of the 15th ACM International Conference on Supercomputing*. Naples, Italy.

Coleman, S. and McKinley, K. S. 1995. Tile size selection using cache organization and data layout. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*. La Jolla, CA, 279–290.

Collard, J.-F. 1994. Space-time transformation of while-loops using speculative execution. In *Proceedings of the Scalable High Performance Computing Conference*. Knoxville, TN, 429–436.

Cormen, T., Leiserson, C., and Rivest, R. 1990. *Introduction to Algorithms*. The MIT Press and McGraw-Hill Book Company.

Ding, C. and Kennedy, K. 2001. Reducing effective bandwidth through compiler ehnancement of global cache reuse. In *Proceedings of International Parallel and Distributed Processing Symposium*.

Ferrante, J., Sarkar, V., and Thrash, W. 1991. On estimating and enhancing cache effectiveness. In *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*. Also in *Lecture Notes in Computer Science*, pp. 328-341, Springer-Verlag, August 1991.

Gary, M. R. and Johnson, D. S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company.

Ghosh, S., Martonosi, M., and Malik, S. 1998. Precise miss analysis for program transformations with caches of arbitrary associativity. In *Proceedings of the Eighth ACM Conference on Architectural Support for Programming Languages and Operating Systems*. San Jose, California, 228–239.

Gu, J., Li, Z., and Lee, G. 1997. Experience with efficient array data flow analysis for array privatization. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Las Vegas, NV, 157–167.

Haghighat, M. R. 1990. Symbolic dependence analysis for high performance parallelizing compilers. Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign.

Hennessy, J. and Patterson, D. 1996. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers.

Jin, G., Mellor-Crummey, J., and Fowler, R. 2001. Increasing temporal locality with skewing and recursive blocking. In *IEEE/ACM SC 2001*. Denver, Colorado.

Kandemir, M., Choudhary, A., Ramanujam, J., and Banerjee, P. 1998. A matrix-based approach to the global locality optimization problem. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*. Paris, France.

Kennedy, K. 2000. Fast greedy weighted fusion. In *Proceedings of the 2000 International Conference on Supercomputing*. Santa Fe, New Mexico.

Kennedy, K. and McKinley, K. S. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Springer-Verlag Lecture Notes in Computer Science, 768. Proceedings of the Sixth Workhsop on Languages and Compilers for Parallel Computing*, Portland, Oregon, August, 1993.

Kodukula, I., Ahmed, N., and Pingali, K. 1997. Data-centric multi-level blocking. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*. Las Vegas, NV, 346–357.

Kodukula, I. and Pingali, K. 1996. Transformations of imperfectly nested loops. In *Proceedings of Supercomputing*.

Lam, M. S., Rothberg, E. E., and Wolf, M. E. 1991. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. Santa Clara, CA, 63–74.

Manjikian, N. and Abdelrahman, T. 1997. Fusion of loops for parallelism and locality. *IEEE Transactions on Parallel and Distributed Systems 8,* 2 (February), 193–209.

Matula, D. and Beck, L. 1981. Smallest-last ordering and clustering and graph coloring algorithms. Tech. Rep. TR CSE 8104, Department of Computer Science and Engineering, Southern Methodist University, Dallas, Texas.

Mitchell, N., Högstedt, K., Carter, L., and Ferrante, J. 1998. Quantifying the multi-level nature of tiling interactions. *International Journal of Parallel Programming 26,* 6 (December), 641–670.

Nguyen, T. and Li, Z. 1998. Interprocedural analysis for loop scheduling and data allocation. *Parallel Computing 24,* 3, 477–504.

Object-Oriented Scientific Computing. *Blitz++*. Object-Oriented Scientific Computing, http://www.oonumerics.org/blitz/benchmarks/.

O'Boyle, M. and Knijnenburg, P. 1997. Non-singular data transformations: Definition, validity and applications. In *Proceedings of ACM International Conference on Supercomputing*. Vienna, Austria, 309–316.

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

Park, N., Hong, B., and Prasanna, V. K. 2002. Analysis of memory hierarchy performance of block data layout. In *Proceedings of International Conference on Parallel Processing*. Vancouver, Canada, 34–44.

Pugh, W. 1992. A practical algorithm for exact array dependence analysis. *Communications of the ACM 35,* 8 (August), 102–114.

Pugh, W. and Rosser, E. 1999. Iteration space slicing for locality. In *Proceedings of the Twelfth International Workshop on Languages and Compilers for Parallel Computing*. San Diego, CA.

Pugh, W., Rosser, E., and Shpeisman, T. 1996. Exploiting monotone convergence functions in parallel programs. Tech. Rep. CS-TR-3636, University of Maryland. October.

Rivera, G. and Tseng, C.-W. 1999. A comparison of compiler tiling algorithms. In *Proceedings of the Eighth International Conference on Compiler Construction*. Amsterdam, The Netherlands.

Rivera, G. and Tseng, C.-W. 2000. Tiling optimizations for 3d scientific computations. In *IEEE/ACM SC 2000*.

Rosser, E. 1998. Fine-grained analysis of array computations. Ph.D. thesis, Department of Computer Science, University of Maryland at College Park.

Sarkar, V. 1998. Loop transformations for hierarchical parallelism and locality. In *Proc. Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*. Springer-Verlag, Pittsburgh, PA.

Song, Y. and Li, Z. 1999. New tiling techniques to improve cache temporal locality. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*. Atlanta, GA, 215–228.

Song, Y., Xu, R., Wang, C., and Li, Z. 2001. Data locality enhancement by memory reduction. In *Proceedings of the 15th ACM International Conference on Supercomputing*. Naples, Italy.

Strout, M., Carter, L., Ferrante, J., and Simon, B. 1998. Schedule-independent storage mapping for loops. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*. San Jose, CA, 24–33.

Temam, O., Fricker, C., and Jalby, W. 1994. Cache interference phenomena. In *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. Nashville, TN, 261–271.

Wise, D. S., Alexander, G. A., Frens, J. D., and Gu, Y. 2001. Language support for morton-order matrices. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Snowbird, Utah.

Wolf, M. 1992. Improving locality and parallelism in nested loops. Ph.D. thesis, Department of Computer Science, Stanford University.

Wolfe, M. 1995. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company.

Wonnacott, D. 2002. Achieving scalable locality with time skewing. *International Journal of Parallel Programming 30,* 3 (June), 181–221.

Xue, J. 2000. *Loop Tiling for Parallelism*. Kluwer Academic Publishers.