# New Tiling Techniques to Improve Cache Temporal Locality *

Yonghong Song   Zhiyuan Li
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
{songyh,li}@cs.purdue.edu

## Abstract

Tiling is a well-known loop transformation to improve temporal locality of nested loops. Current compiler algorithms for tiling are limited to loops which are perfectly nested or can be transformed, in trivial ways, into a perfect nest. This paper presents a number of program transformations to enable tiling for a class of nontrivial imperfectly-nested loops such that cache locality is improved. We define a program model for such loops and develop compiler algorithms for their tiling. We propose to adopt *odd-even variable duplication* to break anti- and output dependences without unduly increasing the working-set size, and to adopt *speculative execution* to enable tiling of loops which may terminate prematurely due to, e.g. convergence tests in iterative algorithms. We have implemented these techniques in a research compiler, Panorama. Initial experiments with several benchmark programs are performed on SGI workstations based on MIPS R5K and R10K processors. Overall, the transformed programs run faster by 9% to 164%.

**Key words** – Caches, loop transformations, optimizing compilers.

## 1 Introduction

Due to the widening gap between processor and memory speed, the importance of efficient use of caches is widely recognized. *Loop tiling*, which combines strip-mining and loop interchange, is a well-known loop transformation which can be used to increase the reuse of cached data [25]. A variety of other loop transformations such as *reversal* and *skewing* have also been proposed to enable loop tiling [23].

Current tiling techniques are generally limited to perfectly-nested loops. Each loop, except the innermost one, in such a perfect nest contains exactly one inner loop at the next level. A loop nest which contains simple statements between two loop headers, as shown in Figure 1, can be transformed into a perfect nest simply by moving those statements, guarded by appropriately inserted IF conditions, inside the inner loop [23], assuming the inner loop iterates at least once.

Unfortunately, many imperfectly-nested loops in numerical programs are not as trivial as the one in Figure 1. A loop may contain several inner loops which are at the same level. For example, the Jacobi program kernel shown in Figure 2(a) has two separate loop nests within the outmost loop. In certain cases, techniques such as *peel-and-fusion* [13] can transform nontrivial imperfectly-nested loops into perfectly-nested ones, which then may become amenable to tiling. Unfortunately, such techniques only exploit temporal locality within the same iteration of the outer loop, e.g., the $T$ loop in the Jacobi code. (In this paper, the $T$-loop is called the *time-step* loop.) Relatively little temporal reuse is exploited because cached data are not reused across different time steps. Current tiling techniques also have difficulties with premature loop exits, such as those caused by convergence tests in iterative numerical algorithms.

This paper presents a number of program transformations to overcome the aforementioned difficulties with tiling. We make two main contributions:

- We define a class of nontrivial imperfectly-nested loops for which we present algorithms to legally and profitably tile the separate loops which are not perfectly-nested in the time-step loop, as in the case of Jacobi. Temporal reuse is thus exploited across different time steps.
- We present a scheme to permit tiling of loops with premature exits by speculatively executing iterations of the time-step loop before the

*This work is sponsored in part by National Science Foundation through grants CCR-950254, MIP-9610379 and by Purdue Research Foundation.

exit condition can be tested for those iterations. Roll-back statements are inserted to insure correct program results in case the exit condition turns out to be true.

The techniques introduced in this paper are particularly suitable for iterative stencil computations in which each matrix element is updated based on the values of the neighboring elements. Such computations are often performed for solving partial differential equations, geometric modeling and image processing. Data dependence information required to automate the proposed techniques can be gathered by existing compiler analyses. We have implemented all the proposed techniques within the Panorama compiler. In order to evaluate their effectiveness, we applied the techniques to one well-known numerical computation kernel, Jacobi, and two application programs, `tomcatv` and `swim`, from the industrial SPEC95 benchmarks [20]. Overall our program transformation improves the execution speed by 9% to 164% on SGI sequential workstations.

```
DO I1 = l1, u1
  Statement S1
  DO I2 = l2, u2
    Statement S2
  END DO
  Statement S3
END DO
```

(a) Before transformation.

```
DO I1 = l1, u1
  DO I2 = l2, u2
    IF (I2 = l2) Statement S1
    Statement S2
    IF (I2 = u2) Statement S3
  END DO
END DO
```

(b) After transformation.

Figure 1: A Trivial Imperfect Loop Nest Transformed into a Perfect Loop Nest.

In this paper, we consider a uniprocessor computer with a memory hierarchy which includes cache memories at one or more levels. We assume that one of the caches is large enough to store a small number of columns of the considered arrays but not the whole arrays. The secondary caches in typical modern workstations satisfy this assumption and the techniques introduced in this paper aim to improve data reuse on such caches.

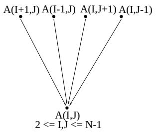
The rest of the paper is organized as follows. In Section 2, we define a class of imperfectly-nested loops and present our tiling scheme to legally tile such loops. We also present our criteria to determine whether our tiling scheme is profitable. We further discuss techniques to improve profitability of our tiling scheme by loop preprocessing. In Section 3, we present our scheme of speculative execution to allow tiling when the time-step loop contains premature exits. In Section 4, we present experimental results. We discuss related work in Section 5 and conclude in Section 6.

## 2 Tiling Imperfectly-Nested Loops

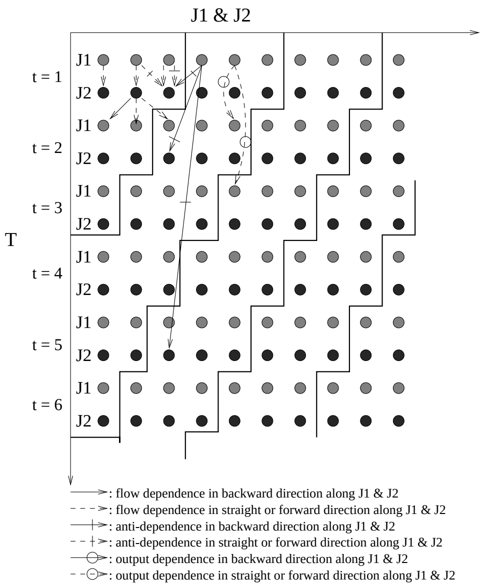
In this section, we first use Jacobi as an example to give a basic idea of our tiling scheme for imperfectly-nested loops. We then define the class of loops to be handled by the scheme. After that, we present the scheme which legally tiles the imperfectly-nested loops and the criteria to determine whether such tiling

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

(a) Jocobi kernel code




(b) Stencil illustration of Jacobi code




(c) Iteration Subspace and Dependences

Figure 2: Code, Computation Stencil, Dependences and Iteration Subspace of Jacobi Kernel.

is profitable. We then discuss techniques to improve the profitability of our tiling scheme by additional loop transformations. After analyzing the complexity of the compiler algorithms, we briefly discuss their extensions.

### 2.1 The Basic Idea

In iterative numerical algorithms, the same array elements are usually accessed repeatedly in different time steps. If the number of array elements is large, then the cache will overflow, which requires the array elements to be fetched repeatedly to the cache in every time step. Using *loop tiling*, before the work in a time step finishes, the execution moves to the next time step to operate on the array elements recently referenced, which avoids cache overflow. The unfinished time step will continue at a later moment. Consider the Jacobi kernel shown in Figure 2(a), which has the convergence test removed to simplify the illustration. We can plot the iteration subspace for the $T$, $J_1$, and $J_2$ loops as shown in Figure 2(c). Within each $T$ iteration, complete execution of $J_1$ and $J_2$ loops requires accesses to nearly $2 \times (N-1)^2$ distinct array elements. With sufficiently large $N$, the cache will overflow, resulting cache misses in the next $T$ iteration. To avoid overflow, we can partition the iteration subspace into *tiles*, as marked by the stair-case lines in Figure 2(c), which are executed in the order from top-left to bottom-right. A properly chosen tile size will make the number of memory locations accessed in each tile fit the cache.

A reordered execution sequence is legal only if it satisfies the original data dependences. Figure 2(b) shows Jacobi's computation stencil. Figure 2(c) shows a subset of data dependences. The edges between the iteration points indicate flow, anti- and output dependences. We adopt a value-based definition of flow dependences [25] such that a *flow dependence* exists from statement $S_1$ to statement $S_2$ if the latter may use the value written by the former. We adopt the traditional definitions of anti- and output dependences [11]. An *anti-dependence* exists from $S_1$ to $S_2$ if the former should read from a memory location before it is overwritten by the latter. An *output dependence* exists from $S_1$ to $S_2$ if $S_2$ may overwrite the memory written by $S_1$. One can similarly define dependences between two variable references, two loop iterations, two program segments and so on [25].

The key to our scheme for tiling imperfectly-nested loops is to find a uniform tile *slope* such that all flow dependences carried by $T$ are satisfied and to find an *offset* for each tiled inner loop such that all flow dependences within the same $T$ are also satisfied. The tile shape, as the result, guarantees that no flow dependences will exist from a later-executed tile to an earlier-executed tile. Anti- and output dependences which exist from later-executed tiles to earlier-executed ones can be eliminated by a technique called *odd-even duplication* of arrays. Unlike full array expansion, odd-even duplication will not unduly increase the working-set size which defeats locality. In Figure 2(c), the edge from node $n_1$ to $n_2$ is drawn solid if the $J$ loop index value for $n_1$ is greater than that of $n_2$, and it is drawn dotted otherwise, where $J$ stands for either $J_1$ or $J_2$. Only the solid edges will affect the legal tile shape. One can see that a solid flow-dependence edge is within the same tile and that a solid anti-dependence edge may cross two neighboring tiles. However, as shown in Section 2.3, the latter can be removed by odd-even duplication. There exist no solid output dependence edges for Jacobi.

## 2.2 A Class of Imperfectly-Nested Loops

Our general program model is presented in Figure 3(a). The $T$-loop body contains $m$ $J$-loops, $m \geq 1$. Each $J$ loop may contain arbitrary program constructs. Without loss of generality, all loops are assumed to have step 1. We require each loop $J_i$ to take the form of $L_i + b_i * T$ and $U_i + b_i * T$ as the lower and upper bounds respectively, where $L_i$ and $U_i$ are $T$-invariants and $b_i$ is a nonnegative known constant. To make the presentation clean, $ITMAX$ is assumed here to be an even number. We require that any read reference gets its value either always from the same $T$-iteration or always from the previous $T$-iteration. Hence, any flow dependence should have the distance of either 0 or 1 at loop level $T$. This is commonly true for the applications of our focus, which use iterative algorithms. Existing compiler techniques for *array privatization* can determine whether such a condition with flow dependences is satisfied (see [5] for a list of references). In the Jacobi code (Figure 2), read references to $A$ always get the values from the previous $T$ iteration except when $T = 1$, which gives the distance 1. Read references to $L$ always get their values from the same $T$ iteration, making the distance 0. In Figure 3(a), we assume that $B$ always gets its value from the previous $T$ iteration.

For simplicity of exposition, in this paper we consider tiling for $J_i$ loops only, $1 \leq i \leq m$, even though they may contain inner loops. The techniques proposed here, however, can be extended to include loops within the $J_i$ loops, which will be briefly discussed in Section 2.6.

```
DO T = 1, ITMAX
  DO J1 =
    L1 + b1 * T,
    U1 + b1 * T
    ...
    ... ← B[...]
    ...
  END DO
  ...
  DO Jm =
    Lm + bm * T,
    Um + bm * T
    ...
    B[...] ← ...
    ...
  END DO
END DO
```

(a) Program Model

```
DO T = 1, ITMAX
  IF (MOD(T, 2).EQ.1) THEN
    DO J1 = L1 + b1 * T, U1 + b1 * T
      ...
      ... ← B[...]
      ...
    END DO
    ...
    DO Jm = Lm + bm * T, Um + bm * T
      ...
      B'[...] ← ...
      ...
    END DO
  ELSE
    DO J1 = L1 + b1 * T, U1 + b1 * T
      ...
      ... ← B'[...]
      ...
    END DO
    ...
    DO Jm = Lm + bm * T, Um + bm * T
      ...
      B[...] ← ...
      ...
    END DO
  END IF
END DO
```

(b) After Odd-Even Array Duplication

```
DO JJ = f1(SLOPE), f2(SLOPE), NSTEP
  DO T = g1(JJ, SLOPE, NSTEP), g2(JJ, SLOPE, NSTEP)
    IF MOD(T, 2).EQ.1) THEN
      DO J1 = L'1, U'1
        ...
        ... ← B[...]
        ...
      END DO
      ...
      DO Jm = L'm, U'm
        ...
        B'[...] ← ...
        ...
      END DO
    ELSE
      DO J1 = L'1, U'1
        ...
        ... ← B'[...]
        ...
      END DO
      ...
      DO Jm = L'm, U'm
        ...
        B[...] ← ...
        ...
      END DO
    END IF
  END DO
END DO
```

(c) After Tiling

Figure 3: Steps of Tiling an Imperfect Loop Nest

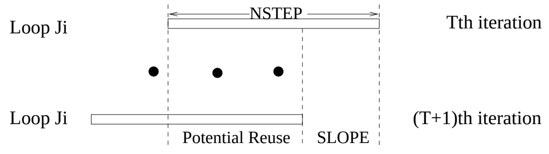



Figure 4: A Portion of A Tile to Illustrate Reuse

- Construct the dependence subgraph.
- Calculate the minimum slope.
- Select the maximum legal tile size and test the profitability.
- Perform array duplication.
- Generate the tiled code.

Figure 5: Main Steps to Tile a Loop Nest

### 2.3 Algorithm for Tiling An Imperfect Loop Nest

Given the loop nest in our program model, our goal is to transform it into the form shown in Figure 3(c), where, by the terms in [23], $JJ$ is called the *tile-controlling loop*, and $J_i$ the *tiled loops*. For convenience, we call loop $T$ the *time-step* loop. The $J_i$ loops are tiled with the size *NSTEP*, and in terms of Figure 2(c), *NSTEP* is the number of grid points for a tile in $J_1$&$J_2$ direction. *SLOPE* is the number of grid points by which the tile shifts to the left when the $T$-index increases by 1. Figure 4 illustrates two instances of loop $J_i$, which shows the shape for a portion of a tile. The potential temporal reuse across $T$ iterations is proportional to the value of $NSTEP - SLOPE$.

Figure 5 shows the main steps in our tiling technique. In the rest of this subsection, we present the compiler algorithms to implement these steps.

#### 2.3.1 Constructing Subgraph

The minimum legal *SLOPE* is constrained by data dependence distances. There exist extensive studies of the problem of computing the data dependence distance between a pair of array references with respect to their common enclosing loops [25]. Where such distance is not a constant, symbolic analysis can be performed to derive bounds on the distance values [3, 16, 7]. In current literatures, dependence distances are usually defined with respect to loops which contain both of the dependent references. For our program model, this would apply to loop $T$ only. We slightly extend the definition to also include the $J_i$ loops.

**Definition 1** In our program model, suppose there exists a dependence from iteration $(T^1, J_i^1)$ to iteration $(T^2, J_k^2)$, we say the dependence has a distance vector of $(T^2 - T^1,\ J_k^2 - J_i^1)$. More specifically, we say the distance with respect to the $< J_i, J_k >$ loops is $J_k^2 - J_i^1$. If $J_k^2 - J_i^1$ is negative, we say there exists a *backward dependence* w.r.t. the $< J_i, J_k >$ loops. □

Existing works on distance-computing cited above can readily be used to compute a lower bound and an upper bound for the distances defined above between references in the same or different $J$ loops. We construct a dependence subgraph [11] for the loop nest in which each node represents a statement within one of the $J_i$ loops. Each edge represents a flow, an anti- or an output dependence. Multiple edges may exist from one node to another. Each edge is marked by the distance vector $(d_T, d_J)$, where $d_T$ is the distance w.r.t. $T$ and $d_J$ is the lower bound estimated for the possible negative distances w.r.t. $J_i$ loops. All the $d_T$ values are grouped into three classes: 0, 1 and any value greater than 1. We will not differentiate different $d_T$ values which are greater than 1, but denote them as $G_1$ equally. Recall that, in our program model, the flow dependence distance with respect to the $T$ loop can only be 0 or 1. Hence, $d_T = G_1$ is possible for anti- and output dependences only. If the distance w.r.t. $J$ cannot be negative, then we let $d_J$ be 0. For the Jacobi example (Figure 2), the dependence subgraph is shown in Figure 6(a), where *S1* is the statement that computes $L$ and *S2* the one that computes $A$. For easy illustration, in Figure 6(a), we draw one edge to represent the same type of data dependences (flow, anti- or output) from one vertex to another, marking all possible dependence distance vectors.

Since loop iterations must be executed in the lexicographical order, any flow dependence edge in the dependence subgraph whose vertices are in the same $J_i$ loop must have a distance vector $(0, 0)$ or $(1, d_J)$, and any anti- or output dependence edge must have a distance vector $(0, 0)$, $(1, d_J)$ or $(G_1, d_J)$. If we have a group of anti- or output dependence edges with the same source, same target and same $d_J$ value, then we delete all but one edge that has the minimum $d_T$ value. The tile shape determined by the remaining dependence edges will also satisfy the deleted ones. Furthermore, we can condense all the nodes (in the dependence subgraph) which belong to the same $J_i$ loop to a single node, resulting in a new graph defined below. This simplified graph filters out dependence information which is unimportant to our tiling scheme, thus improving the efficiency of graph traversal in our technique.

**Definition 2** The *J-loop distance subgraph* $G_l$ is a graph derived from the dependence subgraph $G_d$ of the given loop nest which conforms to the program model such that

1. Each node $J_i$ in $G_l$ represents a $J_i$ loop in the given loop nest.

2. For each flow dependence edge $e = < n_1, n_2 >$ in $G_d$ with the distance vector $(d_T, d_J)$, with $n_1$ in loop $J_{i1}$ and $n_2$ in loop $J_{i2}$, if $J_{i1}$ and $J_{i2}$ are different nodes in $G_l$, add an edge in $G_l$ from $J_{i1}$ to $J_{i2}$ with the distance vector $(d_T, d_J)$. Otherwise, if $J_{i1}$ and $J_{i2}$ are the same and $d_T$ is

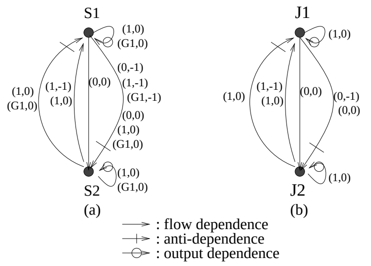



Figure 6: The Dependence and J-loop Distance Subgraphs of Jacobi

equal to 1, add an edge from $J_{i1}$ to $J_{i2}$ with the distance vector $(d_T, d_J)$.

3. For every two nodes $n_1$ and $n_2$ in $G_d$ such that there exist anti- dependence(s) from $n_1$ to $n_2$, with $n_1$ in loop $J_{i1}$ and $n_2$ in loop $J_{i2}$, consider all the anti-dependence edges from $n_1$ to $n_2$.
   (a) Group the anti-dependence edges such that the edges belonging to the same group have the same $d_J$ value.
   (b) For each group, take the minimum $d_T$ value, denoted as $min_{\boldsymbol{iT}}$. Add an edge from $J_{i1}$ to $J_{i2}$ with the distance vector $(min_{\boldsymbol{iT}}, d_J)$ if $min_{\boldsymbol{iT}} \neq G_1$, or $(1, d_J)$ otherwise.
4. Process output dependences in $G_d$ in the same way as in STEP 3.

□
Note that in STEP 3(b), if $min_{\boldsymbol{iT}} = G_1$, we add $(1, d_J)$ to $G_l$ instead of $(min_{\boldsymbol{iT}}, d_J)$. Although such an treatment can potentially produce an over-conservative tile shape, we do not expect $min_{\boldsymbol{iT}} = G_1$ in common cases. For Jacobi's dependence distance subgraph (Figure 6(a)), the $J$-loop distance subgraph is derived as shown in Figure 6(b).

### 2.3.2 Calculating Minimum Slope

The following algorithms perform loop tiling based on the $J$-loop distance subgraph. The first algorithm computes the minimum legal tiling slope, $SLOPE$, and the *offset*s of $J_i$ loop bounds within the same tile.

**Algorithm 1** Compute the Minimum Legal Tiling *Slope* and the *Offset*s.
**Input:** A loop nest, which conforms to our program model, and its $J$-loop distance subgraph $G_l$.
**Output:** $SLOPE$, the minimum legal *slope* for tiling, and the *offset* for each node in $G_l$.
**Procedure:**

- **STEP A:** Compute the *offset* for every node in $G_l$.
  - 1. Temporarily remove all anti- and output dependence edges and any edge with $d_T = 1$.
  - 2. For all nodes in $G_l$ without successors, assign $offset = 0$.
  - 3. If all the nodes in $G_l$ have been assigned *offset* values, go to 6. Otherwise, continue to 4.
  - 4. Find any node $u$ in $G_l$ whose successors $c_i$ $(i = 1, 2, \ldots)$ have all been assigned *offset* values. Each edge is annotated by $(d_{Ti}, d_{Ji})$. Let $offset[u] = max\{|d_{Ji}| + offset[c_i] | i = 1, 2, \ldots\}$.
  - 5. Go back to 3
  - 6. Put all edges removed in Step A.1 back to $G_l$.
- **STEP B:** Let $SLOPE = max\{offset[v] + |d_J| - offset[u] | u, v$ are nodes in $G_l, < u, v >$ is any flow dependence edge with $d_T = 1.$ }

□
For the Jacobi code (Figure 2(a)), after STEP A.1, there exists only one edge from $J_1$ to $J_2$, with the distance vector $(0, 0)$. After initializing $offset[J_2]$ to 0, we have $offset[J_1] = 0$, which means that there is no need to adjust loop bounds for either $J_1$ or $J_2$ within the same $T$ iteration. In STEP B, since there exist only two flow dependence edges (from $J_2$ to $J_1$) with $d_T = 1$, we have $SLOPE = max\{|d_{J_i}| | i = 1, 2\} = 1$.

### 2.3.3 Selecting Tile Size

**Definition 3** We define WSet($x$) as the maximum working set size of any single $T$ iteration in the transformed code (Figure 3(c)), with $NSTEP = x$. □

In any single $T$ iteration of the transformed code, the loop bounds of $J_i$ depend on the combination of $JJ$ and $T$. WSet($x$) takes the maximum working set from those produced by all $JJ$-$T$ combinations. Since the code is yet to be transformed at this compilation stage, we estimate WSet($x$) as $\sum_{i=1}^{m} MemCount_i$, where $MemCount_i$ represents the number of different memory locations accessed in $x$ iterations of loop $J_i$. This estimation gives an upper-bound for WSet($x$). $MemCount_i$ can be estimated during array data flow analysis by merging array regions accessed in each $J_i$ loop body [5]. Next, we determine the size of the tile. The objective is to maximize the tile size in order to maximize data reuse within the tile and to prevent anti- and output dependence from crossing backward more than one tile boundary, but the tile should not be so wide that its working set size exceeds the effective cache size [22].

**Algorithm 2** Profitability Test and Tile Size Determination
**Input:** A loop nest which conforms to our program model, the $J$-loop distance subgraph $G_l$, the $SLOPE$ value computed by Algorithm 1, the given cache size

*CS* and $\alpha$, the *effective-cache factor* [1] [22].
**Output:** The tile size, *NSTEP*, such that tiling is profitable, and a logical value *PROFITABLE*.
**Procedure:**

1. If $L_i + b_i \leq U_i$ holds for some $1 \leq i \leq m$, go to STEP 2. Otherwise $PROFITABLE = NO$ and return from the procedure.

2. Compute $NSTEP = max\{x | (\text{WSet}(x) \leq \alpha * CS)\}$, and then goto STEP 3.

3. If $NSTEP \leq SLOPE$, $PROFITABLE = NO$ and return from the procedure. Otherwise, goto STEP 4.

4. Let $NSTEP1 = max\{offset[v] + |d_J| - offset[u] \; | u, v$ are nodes in $G_l, < u, v >$ is any anti- or output dependence edge with $d_T = 0.\}$, $NSTEP2 = max\{offset[v] + |d_J| - offset[u] - SLOPE \; | u, v$ are nodes in $G_l, < u, v >$ is any anti- or output dependence edge with $d_T = 1.\}$. If $NSTEP < max(NSTEP1, NSTEP2)$, $PROFITABLE = NO$. Otherwise, $PROFITABLE = YES$.

□

STEP 1 guarantees that there is an overlap of iteration points for at least one $J_i$ loop between adjacent $T$ iterations, i.e., $L_i + b_i * 2 < U_i + b_i$ holds for at least one $J_i$ loop. STEP 2 guarantees the working set to fit in the *effective* cache. According to Figure 4, STEP 3 makes *NSTEP* large enough to allow potential temporal reuse across $T$ iterations. STEP 4 prevents any anti- or output dependence from crossing backward more than one tile boundary. In Jacobi's $J$-loop distance subgraph (Figure 6(b)), there exist two anti-dependence edges with $d_T = 0$, which have $d_J = -1$ and $d_J = 0$ respectively. Recall that $offset[J_1] = offset[J_2] = 0$. Hence, $NSTEP1 = max\{|d_J| | d_J = 0, -1\} = 1$. There exist an anti-dependence edge and two output dependence edges with $d_T = 1$ which have $d_J = 0$. Hence, $NSTEP2 = max\{|d_J| - SLOPE | d_J = 0\} = -SLOPE$.

### 2.3.4 Array Duplication

The next algorithm determines whether any particular anti- or output dependence needs to be removed by duplication.

**Algorithm 3** Odd-Even Duplication
**Input:** A loop nest which conforms to our program model, the $J$-loop distance subgraph $G_l$ and data flow upward-exposure information.
**Output:** A loop nest with odd-even array duplication inserted (as shown in Figure 3(b)).
**Procedure:**

- **STEP A:** For every anti- or output dependence edge $e = < J_{i1}, J_{i2} >$ with the distance vector $(d_T, d_J)$, check to see whether (a) $d_T = 0$ and $offset[J_{i2}] + |d_J| \leq offset[J_{i1}]$, or (b) $d_T = 1$ and $offset[J_{i2}] + |d_J| - SLOPE \leq offset[J_{i1}]$. If so, then there is no need for odd-even array duplication, and the procedure returns. Otherwise, continue to STEP B.

- **STEP B:** Duplicate the $T$-loop body and insert an IF-statement such that one copy of the loop body becomes the `THEN` branch and the other copy the `ELSE` branch. The `THEN` branch is executed in odd $T$-iterations and the `ELSE` branch in the even $T$-iterations, respectively.

- **STEP C:** For each anti- or output dependence edge $e = < J_{i1}, J_{i2} >$ with the distance vector $(d_T, d_J)$, if either condition (a) or (b) in STEP A does not hold, identify the array , say $A$, which contributes to this dependence. Declare a copy of $A$.

- **STEP D:** Assume $A$ is an original array, and $C$ its new copy. For each write reference to $A$ in the `THEN` branch, change the referenced array name from $A$ to $C$. For each read reference to $A$, if the value is from an odd $T$-iteration, change $A$ to $C$.

- **STEP E:** Insert proper initialization statements for the new copies of arrays. The regions to be initialized for each newly declared array are upwardly-exposed to the entry of the second $T$ iteration.

□

In the worst case all the arrays could be duplicated. In Jacobi's $J$-loop distance subgraph (Figure 6(b)), the anti-dependence from $J_1$ to $J_2$ with $d_T = 0$ and $d_J = 1$ makes it necessary to duplicate $A$. The array regions $A[1:1, 1:N]$, $A[N:N, 1:N]$, $A[2:N-1, 1:1]$, and $A[2:N-1, N:N]$ are upwardly exposed to the entry of the second $T$ iteration. Their initial values should be copied to the duplicates.

### 2.3.5 Tiled Code Generation

Based on the calculation of $SLOPE$, $NSTEP$ and $offsets$ and the loop transformation performed in Algorithm 3, the following algorithm calculates the loop bounds of the tiled $J_i$ loops, the modified $T$ loop and the new $JJ$ loop in Figure 3(c). It also forms the loop body for each tiled $J_i$ loop.

**Algorithm 4** Tiled Code Generation
**Input:** (1) The $J$-loop distance subgraph, $G_l$, (2) *SLOPE* calculated in Algorithm 1 and *NSTEP* calculated in Algorithm 2, and (3) the loop nest updated in Algorithm 3, which is in the form shown in Figure 3(b).
**Output:** A tiled loop nest.
**Procedure:**

[1] $\alpha \times CS$ is called the *effective cache size*, determined by various system factors, which is recommended by the computer vendor based on empirical locality measurement. We chose $\alpha = 0.3$ in our experiments.

```
DO Ji = max(Li + bi * T, JJ - (T - 1) * SLOPE),
 min(Ui + bi * T, JJ - (T - 1) * SLOPE + NSTEP - 1)
  < BODY_Ji >
END DO
```

(a) $offset = 0$

```
DO Ji = max(Li + bi * T + offset, JJ - (T - 1) * SLOPE + offset),
 min(Ui + bi * T, JJ - (T - 1) * SLOPE + NSTEP - 1 + offset)
  IF (Ji.EQ.(Li + bi * T + offset)) THEN
   DO Ki = Li + bi * T, Li + bi * T + offset - 1
    < BODY_Ki >
   END DO
  END IF
  < BODY_Ji >
END DO
```

(b) $offset > 0$

Figure 7: The Tiled $J_i$ Loop

- For every $J_i$ loop, whose current loop bounds are $L_i + b_i * T$ and $U_i + b_i * T$, respectively, replace the whole loop body by the code segment shown in Figure 7(b) if $\mathit{offset}[J_i] > 0$, in which $< BODY_{J_i} >$ represents the original $J_i$ loop body and $< BODY_{K_i} >$ is identical to $< BODY_{J_i} >$ except that the loop index variable $J_i$ is replaced by $K_i$ in $< BODY_{K_i} >$. If $\mathit{offset}[J_i] = 0$, the $J_i$ loop body is replaced by the code segment shown in Figure 7(a).
- The lower bound of $JJ$ loop in the tiled loop nest is $min\{L_i + b_i | 1 \le i \le m\}$. The upper bound of $JJ$ loop is $max\{U_i + b_i * ITMAX + SLOPE * (ITMAX - 1) | 1 \le i \le m\}$. The lower bound of the new $T$ loop is $max(min\,\{\lceil \frac{JJ - U_i + SLOPE + offset}{b_i + SLOPE} \rceil | 1 \le i \le m\}, 1)$. The upper bound of the new $T$ loop is $min(max\,\{\lfloor \frac{JJ - L_i + SLOPE + offset + NSTEP - 1}{b_i + SLOPE} \rfloor$ $| 1 \le i \le m\}, ITMAX)$.

□

The proof of the correctness of Algorithm 4 is sketched in APPENDIX A. Figure 8 shows the Jacobi code transformed by Algorithm 4. Note that since $\mathit{offset}[J_1] = \mathit{offset}[J_2] = 0$, the loop body in Figure 7(a) applies. The $T$ loop can be further improved, shown in Figure 9, by loop fusion [14, 25] and forward substitution [2], assuming array $L$ is dead at the exit of the whole loop nest. Improvement by such known techniques will not be discussed in details in this paper.

### 2.4 Techniques to Improve Profitability

A number of compiler techniques can be applied to a given loop nest to shorten the backward dependence distances w.r.t. $J_i$ loops and hence to increase the profitability. We present two algorithms in this paper. The first one deals with the problem of *incompatible* $J_i$ loops. Take the example in Figure 10(a). There exists a flow dependence from $J_2$ to $J_1$ such that $d_J = 1 - N$ and hence $SLOPE = N - 1$. We say loops $J_1$ and $J_2$ are incompatible because their indices appear in different dimensions of array $A$. On the other hand, we say loops $J_1$ and $I_2$ are compatible because their

```
DO I = 1, N
 C(I, 1) = A(I, 1)
 C(I, N) = A(I, N)
END DO
DO I = 2, N - 1
 C(1, I) = A(1, I)
 C(N, I) = A(N, I)
END DO
DO JJ = 2, N + ITMAX - 2, NSTEP
 DO T = max(1, JJ - N + 2), min(JJ + NSTEP - 2, ITMAX)
  IF (MOD(T, 2).EQ.1) THEN
   DO J1 = max(2, JJ - T + 1), min(JJ + NSTEP - T, N - 1)
    DO I1 = 2, N - 1
     L(I1, J1) = (A(I1 + 11, J1) + A(I1 - 1, J1) + A(I1, J1 + 1)
        +A(I1, J1 - 1))/4
    END DO
   END DO
   DO J2 = max(2, JJ - T + 1), min(JJ + NSTEP - T, N - 1)
    DO I2 = 2, N - 1
     C(I2, J2) = L(I2, J2)
    END DO
   END DO
  ELSE
   DO J1 = max(2, JJ - T + 1), min(JJ + NSTEP - T, N - 1)
    DO I1 = 2, N - 1
     L(I1, J1) = (C(I1 + 1, J1) + C(I1 - 1, J1) + C(I1, J1 + 1)
        +C(I1, J1 - 1))/4
    END DO
   END DO
   DO J2 = max(2, JJ - T + 1), min(JJ + NSTEP - T, N - 1)
    DO I2 = 2, N - 1
     A(I2, J2) = L(I2, J2)
    END DO
   END DO
  END IF
 END DO
END DO
```

Figure 8: Jacobi Kernel Transformed by Tiling with Odd-Even Duplication

```
...
 DO T = max(1, JJ - N + 2), min(JJ + NSTEP - 2, ITMAX)
  IF (MOD(T, 2).EQ.1) THEN
   DO J = max(2, JJ - T + 1), min(JJ + NSTEP - T, N - 1)
    DO I = 2, N - 1
     C(I, J) = (A(I+1, J)+A(I-1, J)+A(I, J+1)+A(I, J-1))/4
    END DO
   END DO
  ELSE
   DO J = max(2, JJ - T + 1), min(JJ + NSTEP - T, N - 1)
    DO I = 2, N - 1
     A(I, J) = (C(I+1, J)+C(I-1, J)+C(I, J+1)+C(I, J-1))/4
    END DO
   END DO
  END IF
  END DO
...
```

Figure 9: Tiled Jacobi Kernel Improved by Simple Loop Fusion and Forward Substitution

indices appear in the same array dimension. If loops $J_2$ and $I_2$ are permuted as shown in Figure 10(b), then the flow dependence from $I_2$ to $J_1$ has distance vector $(1, 0)$, making the loop nest more profitable to tile. Similarly, loop $I_1$ is said to be compatible with loop $J_2$ and loop $I_1$ and loop $J_1$ can be permuted as shown in Figure 10(c). However, the cache-line spatial locality will suffer due to column-major memory allocation. Array $A$ needs to be transposed [1, 8] in order to restore spatial locality. A formal definition of *compatibility* is given below.

**Definition 4** In the program model in Figure 3(a), suppose each $J_i$ loop contains $s_i$ inner loops, $I_{i,1}$, $I_{i,2}$, ..., $I_{i,s_i}$, $s_i \ge 0$, which are perfectly nested within $J_i$ from the outmost one to the innermost one. Denote $J_i$ by $I_{i,0}$. A *compatible set* is an $(m+1)$-tuple $(c_1, c_2, \ldots, c_m, \mathit{type})$ such that (1) $c_i$ is a nonnegative integer, and $c_i \le s_i$, (2) all loop indices $I_{i,c_i}$, $1 \le i \le m$, should appear in the same dimension of some ar-

```
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)

Figure 10: A Example to Illustrate Compatible Set

rays, and (3) *type* equals $ROW$ (or $COLUMN$) if all the indices appear in the first (or second) dimension. We call this *compatible set* a *row-compatible* (or *column-compatible*) set, respectively. A 2-Dimensional array $A$ is *row-associated* (or *column-associated*) with a row-compatible (or column-compatible) set if at least one loop index variable $I_{i,c_i}$ in that set appears in the first (or second) dimension in some references to $A$. □

The following algorithm checks to see if a combination of array transpose [1, 8] and loop permutation [25] can produce a loop nest which can be profitably tiled. Although there exists an exponential number of possible combinations of permutations of $J_i$ loops and array transpose, only a small number of them need to be considered in practice.

**Algorithm 5** Loop Permutation and Array Transpose to Improve Profitability
**Input:** (1) A loop nest $LP$ which conforms to our program model, (2) *comSets*, all the compatible sets for $LP$, (3) a boolean array $Outmost\_Allowed[1:m][0:s]$ and an array $Perm[1:m][0:s]$ where $s = max\{s_i | 1 \leq i \leq m\}$. $Outmost\_Allowed[u][v]$ indicates whether loop $I_{u,v}$ can become an outmost loop within the $T$ loop after a legal permutation. If $Outmost\_Allowed[u][v] = TRUE$, $Perm[u][v]$ gives one of such legal permutation vectors.
**Output:** A transformed loop nest if the algorithm succeeds or the restored original loop nest if the algorithm aborts.
**Procedure:**
 **if** ($|comSets| > max\{s_i | 1 \leq i \leq m\}$) Abort.
/* Profitable permutations are unlikely to exist */
 **for** (each element $cs = < c_1, c_2, \ldots, c_m, type >$ in *comSets*)
  **if** ($type = ROW$) Perform array transpose on those arrays row-associated with $cs$.
  **if** ($Outmost\_Allowed[i][c_i] = TRUE$ for all $1 \leq i \leq m$)
   Perform the permutation specified by
    $Perm[i][c_i]$, $1 \leq i \leq m$.
   **if** ($PROFITABLE = YES$ after applying Algorithm 2) return the transformed loop nest.
  **end if**
  Undo the permutation and array transpose performed in the above.
 **end for**
 Abort. □

In Jacobi code (Figure 2(a)), where $s_1 = s_2 = 1$, we have two compatible sets, (0, 0, $COLUMN$) and (1, 1, $ROW$). In Figure 10(a), the compatible set (0,1,$COLUMN$) calls for permutation of loop $I_2$ and loop $J_2$ if $Outmost\_Allowed[2][1]$ is $TRUE$. The compatible set (1,0,$ROW$) calls for transposing array $A$ and permutation of $I_1$ and $J_1$ if $Outmost\_Allowed[1][1]$ is $TRUE$.

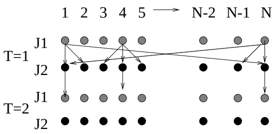
Next, we show that a known loop transformation called *circular loop skewing* [25] can be used to enhance the profitability of our tiling technique. For example, suppose the code skeleton in Figure 11(a) has a wrapped-around computation stencil which is typical for PDEs with circular boundary conditions. Its iteration subspace is shown in Figure 11(b). Assume that, within the same time step $T$, (1) the $j$th iteration of loop $J_2$ is flow-dependent on the $(j-1)$th, $j$th 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, second and last iterations of $J_1$; and (3) the last iteration of loop $J_2$ is flow-dependent on the $(N-1)$th, $N$th and first iterations of $J_1$. Assume that the flow dependence from $J_2$ to $J_1$ has distance vector with $(d_T = 1,\ d_J = 0)$. From Algorithm 1, $offset[J_1] = N-1$, $offset[J_2] = 0$ and $SLOPE = N-1$. Suppose all $J_i$ loops are free of loop-carried dependences. We can reduce $SLOPE$ to 1 as follows. We change $J_2$'s iteration order from $(1, 2, 3, \ldots, N)$ to $(2, 3, \ldots, N, 1)$ for $T = 1$, to $(3, \ldots, N, 1, 2)$ for $T = 2$, and so on. We also change $J_1$'s iteration order from $(1, 2, 3, \ldots, N)$ to $(2, 3, \ldots, N, 1)$ for $T = 2$, to $(3, \ldots, N, 1, 2)$ for $T = 3$, and so on. Such a transformation will eliminate all the backward dependences with $d_J = 1 - N$, resulting in $SLOPE = 1$.

The following algorithm formalizes the circular loop skewing (CLS) technique for profitability enhancement, which circularly skews the bounds of every $J_i$ loop to shorten backward distances. It calls the procedure Find_CLS_Parameter $(LP, G_l, \tau)$, where $\tau < 0$, to eliminate all the backward dependences whose distances are less than $\tau$ in a given loop nest $LP$. Find_CLS_Parameter returns $\delta$, the skewing amount for every $J_i$ loop between adjacent $T$ iterations. It also marks $\beta_i$, $1 \leq i \leq m$, on node $J_i$ in the $J$-loop distance subgraph $G_l$, where $\beta_i$ is the initial skewing amount for each $J_i$ when $T = 1$.

**Algorithm 6** Circular Loop Skewing to Shorten Backward Dependences
**Input:** A loop nest $LP$, which conforms to our program model except that the $J_i$ loop index should have the form of $L_i$ and $U_i$ which are $T$-invariant, its $J$-loop distance subgraph $G_l$, the real cache size $CS$ and the effective cache size factor $\alpha$. Further, every $J_i$ loop is assumed free of loop-carried dependences.
**Output:** A transformed loop nest if successful.
**Procedure:**
 Calculate $NSTEP = max\{x | \mathrm{WSet}(x) \leq \alpha * CS\}$.

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

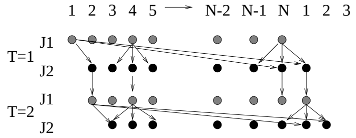
(a)




(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)

Figure 11: A Sample Example Transformed by Algorithm 6: (a) Original Loop Nest Skeleton, (b) Simplified Iteration Space Graph before Transformation, (c) Transformed Loop Nest Skeleton, (d) Simplified Iteration Space Graph after Transformation

$\delta$ = Find_CLS_Parameter $(LP, G_l, 1 - NSTEP)$.
(The code of procedure Find_CLS_Parameter is presented in APPENDIX B.)
**for** (each $J_i$ loop)
Assume the lower and upper loop bounds of $J_i$ are $L_i$ and $U_i$ respectively. Change the loop bounds to $L_i+\beta_i+\delta*(T-1)$ and $U_i+\beta_i+\delta*(T-1)$, and insert an assignment $J_i = MOD(J_i - L_i, U_i - L_i + 1) + L_i$ at the beginning of the $J_i$ loop body.
**end for**
Apply Algorithm 2 to the transformed loop nest.
**if** $(PROFITABLE = YES)$ return the transformed loop nest.
**else** Abort.
**end if** □

Figure 11(c) shows the transformed code for Figure 11(a) after applying Algorithm 6. Figure 11(d) shows its corresponding iteration subspace. Suppose $1 < NSTEP < N$. Procedure Find_CLS_Parameter performs the following: (1) Initialize $\beta_1 = \beta_2 = 0$. (2) Remove all the edges with $d_T = 1$ in $G_l$. (3) For $J_1$, since it does not have any predecessors, do nothing. (4) For $J_2$, increment $\beta_2$ by 1 so the backward dependence distance $1-N$ is eliminated. (5) Restore all the edges with $d_T = 1$ back to $G_l$. (6) After processing the edges with $d_T = 1$, we have $\delta = 1$, $\beta_1 = 0$ and $\beta_2 = 1$.

## 2.5 Complexity Analysis

The construction of the $J$-loop distance subgraph 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 [5, 25] for a list of references). Algorithm 2 is dominated by working-set computation, which requires array region union operations. Such operations are simpler than constructing the $J$-loop distance subgraph.

Given a $J$-loop distance subgraph $G_l = (V, E)$, Algorithm 1 takes $O(|V| + |E|)$ time. In Algorithm 3, finding the arrays to be duplicated needs $O(|V| + |E|)$ time and renaming the references takes time proportional to the loop body size. Algorithm 4 takes time $O(|V| + |E|)$ multiplied by the loop body size. Algorithm 5 theoretically takes $O(\prod_{i=1}^{m} s_i)$ multiplied by the loop body size. ($m$ and $s_i$ are small in practice.) For Algorithm 6, procedure Find_CLS_Parameter has the worst-case time of $O(|V||E|^2)$.

## 2.6 Discussion

Our techniques can be extended to tile the inner loops within $J_i$ loops. Here we take one loop level below $J_i$ as an example. With this extension, each $J_i$ loop has a perfectly nested inner loop, $I_i$, which has lower bound $L'_i + c_i * T$ and upper bound $U'_i + c_i * T$, where $L'_i$ and $U'_i$ are both $T$-invariant and $J_i$-invariant, and $c_i$ is a known nonnegative constant.

The following changes should be made over previous definitions and algorithms.

- Definition 1 needs to be extended such that the distance vector is a triple $(d_T, d_J, d_I)$ instead of a pair $(d_T, d_J)$. Definition 2 should be extended such that each edge in $G_l$ is marked with the new distance vector.
- Algorithm 1 now computes $SLOPE_J$ and $SLOPE_I$ for $J_i$ loops and $I_i$ loops respectively. Also it needs to annotate $G_l$ with $offset_J$ and $offset_I$ for $J_i$ and $I_i$ loops respectively.
- Algorithms 2 now computes $NSTEP_J$ and $NSTEP_I$ for $J_i$ and $I_i$ loops respectively. Definition 3 is extended to define WSet$(x, y)$, $x$ for $NSTEP_J$ and $y$ for $NSTEP_I$. In the new Algorithm 2, we check to see whether WSet$(NSTEP_J, NSTEP_I) \leq \alpha * CS$, where $NSTEP_I = max$ $\{NSTEP_1^I, NSTEP_2^I\}$ and $NSTEP_J = max$ $\{NSTEP_1^J, NSTEP_2^J\}$. If this is true, then we increase $NSTEP_J$ to $max$ $\{x \mid$ WSet$(x, NSTEP_I) \leq \alpha * CS\}$.
- Algorithm 3 now duplicates $A$ if condition (a) or (b) is $TRUE$ for either $J_i$ or $I_i$. Algorithm 4 needs to generate correct loop bounds for $II$, $I_i$, $1 \leq i \leq m$, in addition to $JJ$, $J_i$ and $T$.

## 3 Tiling with Speculative Execution

In numerical programs using iterative methods, there often exist convergence tests which may cause the maximum iteration count not to be reached. To enable our tiling algorithms presented above, we present an algorithm to partition the maximum iteration counts

```
DO T = 1, ITMAX
 DO J1 = L1 + b1 * T, U1 + b1 * T
   ...
 END DO
 ...
 DO Jm = Lm + bm * T, Um + bm * T
   ...
 END DO
 IF (exit condition = TRUE) THEN
  GOTO next
 END IF
END DO
next:
```

(a) Extended Program Model

```
ACCUM = 0
DO CTRL = 1, (ITMAX + LMAX - 1)/LMAX
 A chunk from T = ACCUM + 1 to T = ACCUM + LMAX
  plus exit condition in each T iteration
 ACCUM = ACCUM + LMAX
 IF ( ACCUM.EQ.ITMAX ) THEN goto next
 IF ((ACCUM + LMAX).GT.ITMAX) THEN
  LMAX = ACCUM + LMAX - ITMAX
 END IF
END DO
next:
```

(b) Code with Iteration Chunks

```
ACCUM = 0
Initialize B to A.
DO CTRL = 1, (ITMAX + LMAX - 1)/LMAX
  Execute the tiled chunk from T = (ACCUM + 1)
   to T = (ACCUM + LMAX) .
  ECOND = exit condition for T = (ACCUM + LMAX)
  IF (ECOND.EQ.TRUE) THEN GO TO rollback
  Copy A(or its odd copy C) to backup copy B.
  ACCUM = ACCUM + LMAX
  IF (ACCUM.EQ.ITMAX) THEN GOTO next
  IF ((ACCUM + LMAX).GT.ITMAX) THEN
    LMAX = ACCUM + LMAX - ITMAX
  END IF
END DO
GO TO next
rollback:
  Restore A from backup copy B.
  Execute the original loop nest (Figure 12(a))
   from T = (ACCUM + 1).
next:
```

(c) Tiled and Speculated Code

Figure 12: Tiling with Speculative Execution

into chunks such that the exit condition is tested after the execution of a chunk of $T$-iterations instead of one $T$-iteration. We then tile the individual chunks. In case of overshooting some $T$-iterations, the execution is rolled back. In the speculative code, IF statements must be inserted to guard against potential exceptions such as overflow and divided-by-zero. If a possibility of exceptions is detected, the execution rolls back to the latest checkpoint. The model defined in Section 2.2 is hence extended to include a loop exit test, as shown in Figure 12(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 following algorithm implements the above idea.

**Algorithm 7** Tiling with Speculative Execution
**Input:** A loop nest which conforms to the program model shown in Figure 12(a).
**Output:** A tiled loop nest.
**Procedure:**

- **STEP A:** Estimate the chunk size, $LMAX$.
- **STEP B:** Block the $T$-loop in Figure 12(a) into chunks of size $LMAX$ as shown in Figure 12(b).
- **STEP C:** Tile the chunk using Algorithm 4.
- **STEP D:** Transform the loop nest into one shown in Figure 12(c) where the arrays referenced by exit condition are renamed (which is similar to renaming in Algorithm 3).

□

In the transformed code, for every array $A$ which is the source of $T$-carried flow dependences, we create a *backup copy*, $B$, which is an identical copy of $A$ initially. Variable $ACCUM$ accumulates the total $T$-iterations executed so far. In every iteration of the outmost `DO` loop, after executing the tiled program with $LMAX$ steps, the condition for rollback is checked. If the execution must rollback, the values stored in $B$ are restored and the computation resumes from the beginning of the $(ACCUM+1)$th $T$-iteration, which is the latest checkpoint. Otherwise, $A$ (or its odd copy $C$) is copied to the backup copy $B$. If the execution reaches the $ITMAX$th $T$-iteration, the outmost `DO` loop terminates.

The tradeoff between the over-shooting cost, the checkpointing overhead and the performance gain by tiling can be reasoned as follows. Let $s$ be the original execution time per time step and let $f_{LMAX}$ be the speedup per time step due to tiling, which accounts for checkpointing overhead amortized over $LMAX$ time steps. Assume that originally it takes $T_{total} = p * LMAX + q$ time steps to converge. Since we incur a penalty of $LMAX$ iterations when rollback occurs, tiling with speculation will gain in performance if and only if

$$s * T_{total} > \frac{s}{f_{LMAX}}(p+1) * LMAX + s * q$$

that is,

$$f_{LMAX} > 1 + \frac{LMAX}{T_{total}}$$

We currently use the formula $LMAX = \lceil \frac{NSTEP}{SLOPE} \rceil$, which equals the estimated time steps per tile, given tile size $NSTEP$. (In future work, one can certainly experiment with other formulas. For instance, $LMAX$ may vary at distinct computation stages.) In Section 4 we can see that the gain from cache temporal reuse outweighs the loss due to checkpointing and over-shooting.

## 4 Experimental Evaluation

We have implemented our techniques in Panorama compiler [5]. We apply our algorithms to one well-known numerical kernel, Jacobi with convergence test and two SPEC95 benchmarks `tomcatv` and `swim`, which run on two SGI uniprocessor workstations, one based on a MIPS R5K processor and the other based on a MIPS R10K. The R5K processor has a 32KB 2-way L1 data cache and 512KB 2-way unified L2 cache. The R10K processor also has a 32KB 2-way L1 data cache, but a 2MB 2-way unified L2 cache. Moreover, the MIPS R10K performs out-of-order instruction execution, provides data prefetching instructions, and permits multiple pending cache misses. The native compiler is MIPSPro F77 compiler. In all the experiments on the original code and on the peel-and-fused code, we turn on the "-O3" switch which enables a number of loop transformations including interchange, fusion, fission and tiling. For the code generated by our techniques, "-O2" and "-O3" delivers nearly the same performance. (we show "-O2" results here). The original `tomcatv` and `swim` programs fail the profitability test. We apply Algorithms 5 and 6 to these two programs before taking the general steps shown in Figure 5. Algorithm 5 finds that `tomcatv` can be made profitable for tiling by loop permutation and array transpose. Algorithm 6 finds that `swim` can be made profitable by circular loop skewing. Algorithm 7 applies to Jacobi and `tomcatv`. (The `swim` program does not have exit conditions in the time-step loop.) Jacobi code is also further optimized by forward substitution (Figure 9).

### The Jacobi Program

We fixed $ITMAX$ to 100 and vary the input matrix size arbitrarily. Based on Algorithm 2, $NSTEP$ is derived as 10, 9 and 7 respectively for $N = 869$, 1024 and 1279 on the R5K-based workstation. For the same matrix sizes on the R10K-based workstations, $NSTEP$ is derived as 44, 37 and 30 respectively. Table 1 shows the performance results for the original program, peel-and-fusion and the program tiled with Algorithm 4, marked as "Tiling w/ Array Dup.". Although based on our tiling technique, array $A$ is duplicated, as shown in Figure 8, array $L$ is eliminated after forward substitution (Figure 9), giving no increase of memory usage.

For the peel-and-fusion scheme, we manually apply the methods presented in LRW [12] and TSS [6] to choose the tile size with both the real cache size and *effective cache size* [22]. The best of these four schemes, which gives shortest execution time, is chosen to evaluate peel-and-fusion. From Table 1, tiling with odd-even duplication always performs better than peel-and-fusion. It improves the performance by 70% to 84% in R5K and by 159% to 164% in the R10K over the original code.

### The `tomcatv` Program

Among the 7 $N$-by-$N$ arrays in `tomcatv`, two arrays, $X$ and $Y$, are duplicated, increasing the memory usage by 29%. $NSTEP$ equals 5 and 21 for the R5K- and the R10K-based workstations respectively. With the reference input data, `tomcatv` always runs to the maximum time step, i.e., 750. In order to measure the potential overhead of rollback, we alter the convergence test to trigger rollback for comparison. Different convergence tests result in different number of $T$-iterations executed, both for the untransformed and the transformed programs. The execution time of different versions of `tomcatv` on the R5K is shown in Table 2, where "Iter. Orig." means the number of $T$ iterations executed in the untransformed code, and "Iter. Trans." means the number of $T$ iterations executed in the transformed code before rolling back. The chunk size $LMAX$ is 5 based on our heuristic. When rollback happens we always incur a penalty of $LMAX$ iterations. However, except for those cases which converge very fast, we still get steady speedup which is up to 1.28. When there is no rollback (the $T$-loop bound 750 is reached), the speedup is 1.69.

Table 2: Execution Time (in Seconds) of `tomcatv` on R5K with Various Convergence Tests

| Iter. Orig. | Iter.Trans. | Exec. time | Orig. Time | Speedup |
|---|---|---|---|---|
| 11 | 16 | 25 | 26 | 1.04 |
| 38 | 43 | 47 | 52 | 1.11 |
| 58 | 63 | 63 | 71 | 1.13 |
| 78 | 83 | 79 | 90 | 1.14 |
| 112 | 117 | 105 | 122 | 1.18 |
| 159 | 164 | 143 | 169 | 1.18 |
| 179 | 183 | 158 | 187 | 1.18 |
| 199 | 204 | 175 | 207 | 1.18 |
| 731 | 736 | 593 | 717 | 1.28 |
| 750 | 750 | 434 | 732 | 1.69 |

Table 3: Execution Time(in Seconds) of `swim`

| Test Program | R10K | | R5K | |
|---|---|---|---|---|
| | Exec. Time | Speedup | Exec. Time | Speedup |
| Orig. Prog. | 326.85 | 1.00 | 625 | 1.00 |
| Trans. Prog. | 215.01 | 1.52 | 573 | 1.09 |

On the R10K the original program executes for 366 seconds. The transformed program takes 288 seconds, a speedup of 1.27. With the convergence test altered, the performance comparison is similar to that on the R5K. We also run `tomcatv` with peel-and-fusion, with which the transformed code takes 649 and 306 seconds on the R5K and the R10K and achieves a speedup of 1.13 and 1.19 respectively over the original program.

### The `swim` Program

No arrays are duplicated for `swim`. $NSTEP$ equals 5 and 22 for the R5K and the R10K respectively. The performance results are shown in Table 3, where "Orig. Prog." stands for the original program, and "Trans. Prog." represents the transformed program. The R5K has a smaller L2 cache and therefore a smaller $NSTEP$, which results in less overlap of iteration points between adjacent $T$ iterations within a tile, thus yielding a lower speedup. Peel-and-fusion does not apply directly to `swim` due to long backward dependence distances.

On the R10K, in addition to execution time, we also measure the secondary cache misses using *perfex* library based on R10K performance counters. Table 4 shows the result where 'LS' stands for the number of dynamic load-store instructions, 'SM' for the number of misses for the secondary cache, and 'MR' for the secondary cache miss ratio. In Jacobi, even with checkpointing, the number of dynamic load-store instructions are reduced due to forward substitution.

Table 1: Execution Time (in Seconds) of Different Versions of Jacobi

| Different Scheme | | Matrix Size for R5K | | | Matrix Size for R10K | | |
|---|---|---|---|---|---|---|---|
| | | 869 | 1024 | 1279 | 869 | 1024 | 1279 |
| Original | Time | 46 | 68 | 102 | 24.43 | 44.65 | 52.93 |
| | Speedup | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| Peel-and-fusion | Time | 37 | 50 | 82 | 17.08 | 27.23 | 35.76 |
| | Speedup | 1.24 | 1.36 | 1.24 | 1.43 | 1.64 | 1.48 |
| Tiling w/ Array Dup. | Time | 25 | 38 | 60 | 9.25 | 17.23 | 20.14 |
| | Speedup | 1.84 | 1.79 | 1.70 | 2.64 | 2.59 | 2.63 |

We believe that the difference in the number of load-store instructions between the original code and the peel-and-fusion code is due to the slightly more memory optimization opportunities for the native compiler after peel-and-fusion. For `tomcatv`, the number of load-store instructions is increased by our technique due to periodical checkpointing. In `swim`, the added number of load-store instructions is from the expansion of some arrays to make the program conforming to our program model. In all five cases the secondary cache miss rate is dramatically reduced. The gain in temporal locality across $T$ iterations overcomes the loss of more memory instructions, thus improving the overall performance.

During the experiments, we applied inter-variable padding (similar to the scheme in [18]) to the tiled `swim`. This padding reduces cache set conflicts, hence improving speedup from 1.08 to 1.52. We also applied inter-variable padding to all three original programs, but without noticeable performance gain. This is because cache overflow in the original program dominates the performance.

## 5 Related Work

Kodukula *et al.* propose *data shackling* [9], which blocks the arrays based on data flow analysis and then forms new loop nests to compute block by block. Although it can tackle 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.

Manjikian *et al.* present peel-and-fusion [13]. Their method can partially fuse the adjacent loops within the same $T$ loop and tile the fused loops, but it only exploits locality within the same $T$ loop iteration. Our Algorithm 1 bears some similarity to their peel-factor computation, but they consider dependences within the same $T$-iteration only, while we further consider dependences across $T$-iterations.

Kodukula and Pingali propose a matrix-based framework to represent transformations of imperfectly-nested loops [10] including permutation, reversal, skewing, scaling, alignment, distribution and jamming. Their work does not apply to tiling.

Strout *et al.* discusses the minimum storage requirement to allow flexible loop scheduling such as tiling [21]. Their method does not deal with the imperfectly nested loops handled by our work.

Collard proposes a method to speculatively execute while-loops on parallel machines [4]. His objective is to increase parallel process 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 [17]. Both works by Collard and by Pugh *et al.* present interesting ideas to handle loop exit conditions, which may be incorporated in our future work.

McCalpin and Wonnacott develop a scheme called *time skewing*, which adopts a value-based flow analysis to optimize for memory locality [15]. Their method first performs full array expansion and forward substitution, and then recompresses the expanded array while preserving data dependences. Their method handles a subset of the imperfectly-nested loops represented by our program model.

## 6 Conclusion and Future Work

In this paper, we have presented new techniques to tile nontrivial imperfectly-nested loops. We handle a class of imperfectly-nested loops which seem quite general in scientific computing. We develop two algorithms to shorten backward dependence distances so as to improve profitability. We also extend our program model to handle exit conditions, which are common in iterative algorithms. We implement our techniques in a Fortran source-to-source compiler, Panorama [5]. Preliminary experimental results show the transformed programs run faster by 9% to 164% than those only optimized by the native compiler.

This work has opened several interesting subjects for our future work. The most important ones include the application of the idea of speculative execution to other locality-enhancement techniques and a study of the effect of aggressive locality-enhancement algorithms on the effective cache size.

## Acknowledgement

We thank Chau-Wen Tseng for his careful review before this paper is finalized.

## References

[1] J. M. Anderson, S. P. Amarasinghe and M. S. Lam. Data and computation transformations for multiprocessors. In *Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, July 19-21, 1995.

Table 4: Secondary Cache Utilization of Different Versions of Jacobi, `tomcatv` and `swim`
(LS and SM are counters in millions)

| Test Program | Original | | | Peel-and-Fusion | | | Tiled | | |
|---|---|---|---|---|---|---|---|---|---|
| | LS | SM | MR | LS | SM | MR | LS | SM | MR |
| Jacobi (N = 879) | 473.3 | 19.4 | 0.040 | 465.9 | 9.8 | 0.020 | 401.7 | 0.6 | 0.0015 |
| Jacobi (N = 1024) | 673.9 | 26.1 | 0.039 | 669.6 | 13.4 | 0.020 | 588.5 | 0.9 | 0.0015 |
| Jacobi (N = 1279) | 1003.9 | 40.7 | 0.041 | 1002.5 | 20.5 | 0.020 | 821.2 | 1.9 | 0.0023 |
| `tomcatv` | 12272.0 | 245.7 | 0.020 | 11681.2 | 136.9 | 0.012 | 12775.5 | 111.8 | 0.0088 |
| `swim` | 11401.4 | 866.3 | 0.076 | - | - | - | 12287.3 | 9.9 | 0.0008 |


[2] David F. Bacon, Susan L. Graham and Oliver J. Sharp. Compiler transformations for high-performance computing. In ACM Computing Surveys, Vol. 26, No. 4, Dec. 1994.

[3] W. Blume and R. Eigenmann. Symbolic range propagation. *Proceedings of the 9th International Parallel Processing Symposium*, April 1995.

[4] Jean-Francios Collard. Space-time transformation of while-loops using speculative execution. In *Proc. of the Scalable High-Performance Computing Conf.*, Knoxville, TN, pp. 429-436, May 1994.

[5] J. Gu, Z. Li, and G. Lee. Experience with efficient array data flow analysis for array privatization. In *Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM Press, June 1997.

[6] Stephanie Coleman and Kathryn S. McKinley. Tile size selection using cache organization and data layout. In *Proc. of the ACM SIGPLAN conference on Programming Language Design and Implementation*, June 1995.

[7] M. R. Haghighat. *Symbolic Dependence Analysis for High Performance Parallelizing Compilers*. Ph.D. thesis, CSRD Rpt No. 995, University of Illinois, May 1990.

[8] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. A matrix-based approach to the global locality optimization problem. In *Proc. International Convergence on Parallel Architectures and Compilation Techniques (PACT'98)*, October 14-17,1998, Paris, France.

[9] Induprakas Kodukula, Nawaaz Ahmed and Keshav Pingali. Data-centric multi-level blocking. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun 1997.

[10] Induprakas Kodukula, Keshav Pingali. Transformations of imperfectly nested loops. In *Proc. Supercomputing*, November 1996.

[11] D. J. Kuck. *The Structure of Computers and Computations*, Volume 1. John Wiley & Sons, 1978.

[12] Monica S. Lam, Edward E. Rothberg and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proc. of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63-74, Santa Clara, California, April 8-11, 1991.

[13] Naraig Manjikian and Tarek S. Abdelrahman. Fusion of loops for parallelism and locality. In *IEEE Transactions on Parallel and Distributed Systems*,Vol. 8, No. 2, Feb 1997.

[14] Karhryn S. McKinley, Steve Carr and Chau-Wen Tseng. Improving data locality with loop transformations. In *ACM Transactions on Programming Languages and Systems*, Vol. 18, No. 4, pp. 424–453, July 1996.

[15] John McCalpin and David Wonnacott. Time Skewing: A Value-Based Approach to Optimizing for Memory Locality. In *http://www.haverford.edu/cmsc/davew/cache-opt/cache-opt.html*.

[16] William Pugh. A Practical Algorithm for Exact Array Dependence Analysis. In *Communications of the ACM*, August, 1992.

[17] W. Pugh, E. Rosser and T. Shpeisman. Exploiting Monotone Convergence Functions in Parallel Programs. Technical Report CS-TR-3636.1, University of Maryland, October 1996.

[18] Gabriel Rivera and Chau-Wen Tseng. Eliminating Conflict Misses for High Performance Architectures. In *Proc. of the 1998 ACM International Conference on Supercomputing*, Melbourne, Australia, July 1998.

[19] B. R. Rau and J. A. Fisher. Instruction-level parallel processing: History, overview and perspective. *The Journal of Supercomputing*, 7:9-50, 1993.

[20] Standard Performance Evaluation Corporation, *SPEC Newsletter*, Vols. 1–9, 1989–1997.

[21] Michelle Strout, Larry Carter, Jeanne Ferrante and Beth Simon. Schedule-independent storage mapping for loops. In *Prof. of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October 1998.

[22] Michael E. Wolf, Dror E. Maydan and Ding-Kai Chen. Combining loop transformations considering caches and scheduling. In *MICRO 29*, pages 274-286, Mountain View, CA, 1996.

[23] Michael E. Wolf. *Improving Locality and Parallelism in Nested Loops*. Ph.D. thesis, Stanford University, Aug. 1992.

[24] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proc. of ACM SIGPLAN conference on Programming Language Design and Implementation*, June 1991.

[25] Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1995.


## APPENDIX A

**Theorem 1** Algorithm 4 preserves the semantics of the input loop nest.

**Sketch of Proof:**

**STEP A: Consider all the flow dependences.**

**1. Consider flow dependences within the same $T$ iteration, i.e., $d_T = 0$.** Take an arbitrary flow dependence edge $< u, v >$ with the distance vector $(d_T, d_J)$ where $d_T = 0$. If $u$ and $v$ are identical, such a dependence will be automatically satisfied by the

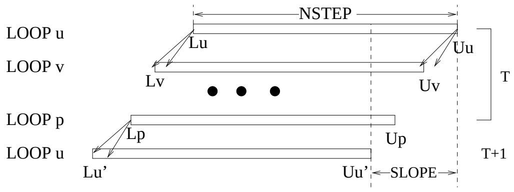



Figure 13: A Portion of One Tile: $L_i$ and $U_i$, $i = u, v$ or $p$, represent the lower and upper bounds of the loops $u$ $v$ and $p$ in the $T$th iteration respectively, and $L'_u$ and $U'_u$ stand for the lower and upper bounds of the loop $u$ in the $(T+1)$th iteration.

forward execution order of the tiling controlled loops. If $u$ and $v$ are not identical, as illustrated in Figure 13, $< u, v >$ exists within the same tile or from an earlier-tile to a later-tile if and only if

$$L_v - L_u \leq d_J \tag{1}$$

is satisfied. From Algorithm 4, Inequality (1) above is equivalent to

$$offset[v] + |d_J| \leq offset[u] \tag{2}$$

which is implied by STEP A of Algorithm 1. The equivalence of Inequalities (1) and (2) is clear for non-boundary case where $L_v = JJ - (T-1) * SLOPE + offset[v]$ and $L_u = JJ - (T-1) * SLOPE + offset[u]$ (Figure 7(b)). For boundary case, similar arguments can prove its equivalence.

**2. Consider flow dependences across different $T$ iterations, i.e., $d_T = 1$.** Take an arbitrary flow dependence edge $< p, u >$ with the distance vector $(d_T, d_J)$ where $d_T = 1$ as in Figure 13. $< p, u >$ exists within the same tile or from an earlier-tile to a later-tile if and only if

$$L'_u - L_p \leq d_J \tag{3}$$

is satisfied. From Algorithm 4, Inequality (3) above is equivalent to

$$offset[u] + |d_J| - offset[p] \leq SLOPE \tag{4}$$

which is implied by STEP B of Algorithm 1. The proof of equivalence is similar to STEP A.1.

**STEP B: *NSTEP*, chosen in Algorithm 2, prevents any anti- or output dependence edge from crossing backward more than one tile boundary, and the criteria in Algorithm 3 to choose which arrays need duplicating is correct.**
The proof is similar to STEP A.

**STEP C: Each instance of $J_i$ $(i = 1, \ldots, m)$ loops in the original loop will be executed exactly once in the transformed code.**

We omit the proof of STEP B and STEP C due to space limit. In conclusion, Algorithm 4 preserves the semantics of the given loop nest. □

## APPENDIX B

**Procedure** Find_CLS_Parameter($LP, G_l, \tau$)
**Input:** (1) The input loop nest $LP$ where each $J$ loop has the lower bound $L_i$ and upper bound $U_i$ which are $T$-invariants. (2) The $J$-loop distance subgraph, $G_l$, for $LP$. (3) A threshold value $\tau$.
**Output:** $\delta$, the skewing amount for every $J_i$ loop between adjacent $T$ iterations.
**Side Effect:** A revised $G_l$.
**Body:**
Temporarily remove all edges with $d_T = 1$ in $G_l$.
Initialize $\beta_j = 0$ for all $J_i$ loops in $G_l$.
**for** (each $J_i$ in $G_l$ following a topological sort)
Denote all of $J_i$'s predecessors as $c_1, c_2, \ldots, c_p$, and the corresponding $\beta$ values as $r_1, r_2, \ldots, r_p$.
**for** $k = 1, p$
**if** ($d_J \geq \tau$ holds for all dependence vectors $(d_T, d_J)$ from $c_k$ to $J_i$)
Skip to the next $k$ iteration.
**end if**
Choose $\eta = max\{d_J | d_J < \tau, (d_T, d_J)$ is a dependence vector for some edge from $c_k$ to $J_i$.$\}$
Assuming the lower and upper loop bounds of $J_i$ are $L_i$ and $U_i$ respectively, and the loop bounds of $c_k$ are $L_r$ and $U_r$ respectively, let $\kappa = U_r + \beta_{c_k} + \eta - L_i - \beta_{J_i} + 1$. Adjust $\beta_{J_i}$ by $\beta_{J_i} = \beta_{J_i} + \kappa$. Remove all the dependence edges along $< c_k, J_i >$ which have its $J$-loop dependence distance less than or equal to $\eta$.
Do depth first search rooted at $J_i$ in $G_l$. Consider each node $J_c$ in the path of depth first search. Suppose $J_c$ has the lower bound $L_c$ and the upper bound $U_c$. We classify $J_c$'s parent $J_p$, which has the lower bound $L_p$ and the upper bound $U_p$, into two classes: (1). If $J_p$ is in the subtree rooted at $J_i$, simply let $\beta_{J_c} = \beta_{J_c} + \kappa$. (2). Otherwise, we eliminate all the dependence vectors $(d_T, d_J)$ from $J_p$ to $J_c$ such that $d_J < (L_c + \beta_{J_c} - U_p - \beta_{J_p} + \kappa)$, then we assign $\beta_{J_c} = \beta_{J_c} + \kappa$.
**end for**
**end for**
Restore all the edges in $G_l$ which were removed in the beginning of the procedure.
$\delta = 0$
**for** (each edge $e = < J_u, J_v >$ with $d_T = 1$)
Assume that the lower and upper bounds of loop $J_u$ are $L_u$ and $U_u$ respectively, and those of $J_v$ are $L_v$ and $U_v$ respectively. Assume the dependence vector for $e$ to be $< d_T, d_J >$.
**if** ($d_J \geq \tau$)
$\delta = max(\delta, \beta_{J_u} - \beta_{J_v})$
else
$\delta = max(\delta, \beta_{J_u} - \beta_{J_v} + U_u + d_J - L_v + 1)$
**end if**
**end for** □