A Compiler Framework for Tiling Imperfectly-Nested Loops *

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

Abstract

This paper presents an integrated compiler framework for tiling a class of nontrivial imperfectly-nested loops such that cache locality is improved. We develop a new memory cost model to analyze data reuse in terms of both the cache and the TLB, based on which we compute the tile size with or without array duplication. We determine whether to duplicate arrays for tiling by comparing the respective *exploited reuse factors*. We identify *compatible loops* in order to improve the profitability of tiling. The preliminary results with several benchmark programs show that the transformed programs run faster by 9% to 282%.

1 Introduction

This paper considers loop tiling [17] as a technique to improve data locality. Previous tiling techniques are generally limited to perfectly-nested loops. Unfortunately, many important loops in reality are imperfectly nested. In our recent work [12], we define a class of imperfectly-nested loops, and present a set of algorithms to tile such loops with *odd-even array duplication*. In this paper, we make the following new contributions:

- We present an integrated compiler framework for tiling the class of nontrivial imperfectly-nested loops considered in [12] such that data locality is improved for both the cache and the TLB.
- We develop a new memory cost model to analyze data reuse in terms of both the cache and the TLB, based on which we compute the tile size with or without array duplication. We determine whether to duplicate arrays for tiling by comparing the respective *exploited reuse factors*. We identify *compatible loops* in order to improve the profitability of tiling.

We implemented our basic tiling algorithms [12] within Panorama, a Fortran source-to-source research compiler [4]. We are implementing the new techniques presented in this paper. In order to pre-evaluate the effectiveness of the new techniques, we manually applied them to two well-known numerical computation kernels, Jacobi and Livermore Loop No. 18 (LL18), and two programs, `tomcatv` and `swim`, from the industrial SPEC95 benchmarks [13]. Overall, our program transformation improves the execution speed of the original programs by 9% to 282% on one SGI uniprocessor workstation and one processor on an SGI multiprocessor machine. These results motivate further experiments on multiprocessor workstations in our future work.

In this paper, we focus on enhancing data locality on a uniprocessor computer with a memory hierarchy which includes an address translation look-aside buffer (TLB) and 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

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

arrays but not the whole arrays. The secondary caches in typical modern workstations satisfy this assumption. The techniques introduced in this paper aim to improve data reuse on such caches and TLB.

The rest of the paper is organized as follows. In Section 2, we define our program model and review basic concepts in our tiling techniques. We present our memory cost model and a tile-size selection scheme in Section 3. Section 4 discusses how to identify *compatible loops*. Section 5 compares tiling with and without duplication. Section 6 presents our integrated framework and a running example. We then evaluate our framework in Section 7. Related work is presented in Section 8, followed by a conclusion in Section 9.

2 Preliminaries

2.1 Program Model

Our program model [12] is shown in Figure 1(a). The T-loop body contains m loops, J_1, J_2, …, J_m ($m \geq 1$), at the next inner level. We call these loops the J loops. 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. To simplify the computation of the distances of value-based flow dependences, we require that any flow dependence should have the distance of either 0 or 1 at the loop level T. This is commonly true for the applications of our focus, which repeatedly access the same arrays. Existing compiler techniques for *array privatization* can determine whether such a condition on flow dependences is satisfied (see [4] for a list of references). In the Jacobi code (Figure 2(a)), whose computation stencil is shown in Figure 2(b), the flow dependences for array A have a distance of 1 at the loop level T, while the flow dependence for array L has distance 0. In Figure 1(a), we assume that B always gets its value from the previous T iteration, i.e., with flow dependence distance of 1.

For simplicity of exposition, in this paper we consider tiling for the J loops only, even though they may contain inner loops. The techniques proposed here, however, can be extended to include loops within the J loops, as in [12]. Several known simple transformations, such as code sinking [12, 17] and creation of single-iteration loops, can be used to transform an imperfectly-nested loop to conform to our program model.

2.2 Basic Concepts

In [12], we develop a set of compiler algorithms to tile the loops conforming to our program model (Figure 1(a)). The key to our scheme 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 iteration 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. For our program model, Figure 1(b) shows the code shape after odd-even duplication, but before tiled-code generation, where B' is a copy of B. Figure 1(c) shows the final tiled code, where *SLOPE* represents the uniform slope angle, *NSTEP* stands for the tile size. We call *JJ* the *tile-controlling loop* and T the *tile-sweeping loop*. Figure 2(c) shows the iteration subspace and dependences for Jacobi, where the shaded rectangle is called a *tile* and the area between two stair-case lines is called a *tile traversal* in this paper. Two tiles are called *consecutive* if they belong to the same tile traversal and the difference of their corresponding T values is 1. The slope angle, *SLOPE*, equals to the maximum number of J loop iterations left-shifted between two consecutive tiles. While *SLOPE* determines the relative position between different tiles, the *offset* value characterizes the relative position between different J loops within the same tile. In Figure 2(c), *SLOPE* equals 1, *NSTEP* equals 3, and the *offset*s are equal to 0 for both loops J_1 and J_2. After tiling (Figure 1(c)), each *JJ* iteration defines one tile traversal.

```
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 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 with Duplication

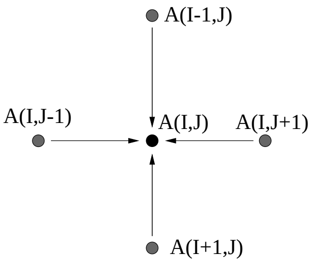
```
DO JJ = f3(SLOPE),
  f4(SLOPE), NSTEP
  DO T =
    g3(JJ, SLOPE, NSTEP),
    g4(JJ, SLOPE, NSTEP)
    DO J1 = L''1, U''1
      ...
      ... ← B[...]
      ...
    END DO
    ...
    DO Jm = L''m, U''m
      ...
      B[...] ← ...
      ...
    END DO
  END DO
END DO
```

(d) After Tiling without Duplication

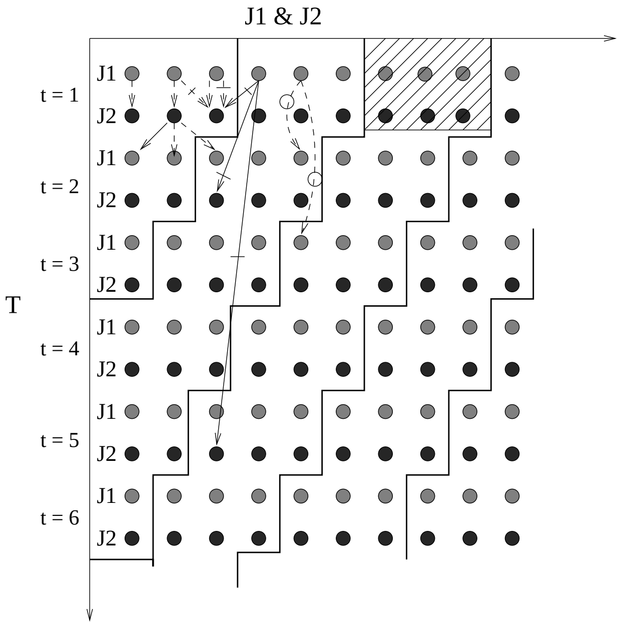
Figure 1: Program Model and Various Intermediate Code Shapes during Transformation

```
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

: flow dependence in backward direction along J1 & J2
: flow dependence in straight or forward direction along J1 & J2
: anti-dependence in backward direction along J1 & J2
: anti-dependence in straight or forward direction along J1 & J2
: output dependence in backward direction along J1 & J2
: output dependence in straight or forward direction along J1 & J2

Figure 2: Code and Computation Stencil of Jacobi Kernel

We use a *J-loop distance subgraph* to represent dependence relations between J loops, where each J loop is represented as a node in the graph [12]. Long backward dependence distances make the loop less profitable to tile. In certain cases, we can permute loops, possibly in conjunction with array transpose, to achieve *compatibility* [12] among the J loops, so as to shorten dependence distances.

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

In Jacobi (Figure 2(a)), $(1, 1, COLUMN)$ and $(2, 2, ROW)$ are two compatible sets [1].

[1] To be more illustrative, in the rest of paper we replace c_i in the compatible set by its index variable names. For example, two compatible sets in Jacobi would be (J_1, J_2, $COLUMN$) and (I_1, I_2, ROW).

Table 1: Parameters of the Cache and the TLB

Notation	C_s	C_b	T_s	T_b
Description	Cache size	Cache block size	TLB size	TLB block size

3 A Memory Cost Model and Tile-Size Selection

In our previous work [12], we assume a fully-associative cache and compute the tile size to maximize cache utilization only. In this paper, we still consider a fully-associative cache, but refine our model by a detailed analysis of cache misses and TLB misses.

Different levels of reuse [15] exist in our program model: *reuse carried by* T, *reuse between different* J *loops*, *reuse carried by a particular* J *loop* and *reuse within one iteration of a* J *loop*. The goal of this paper is to maximize the reuse carried by T. Note that our method improves the temporal locality without sacrificing the original spatial locality in the innermost loop, since the access order in the innermost loop is not changed.

If the tile size *NSTEP* is chosen properly, the tiled loop nest will have the following properties:

- **Property 1:** Any repeated reference to the same memory location within the same tile should be a cache hit and a TLB hit.
- **Property 2:** The overlapping data [2] between two consecutive tiles should be kept within the cache and the TLB when the execution proceeds from one tile to the next.

(Later on in this section, we will discuss how to preserve these two properties.)

Next, we construct a memory cost model to characterize the cache and the TLB and we derive a scheme to select the optimal tile size for our loop transformation, assuming that Properties 1 and 2 hold. Table 1 lists the key sizes measured in the number of data elements. We assume the LRU replacement policy both for the cache and for the fully-associative TLB. Let $\gamma_1 = min\{L_i | 1 \leq i \leq m\}$ and $\gamma_2 = max\{U_i | 1 \leq i \leq m\}$ represent the minimum and the maximum of J_i index values in the original loop nest. Let $\gamma = \gamma_2 - \gamma_1 + 1$, which is the length of the range of all legal J_i loop index values. For simplicity, we assume the size of the data set accessed within each T iteration is T-invariant [3], denoted by W. We make three assumptions in our estimation of the cache and TLB misses:

- **Assumption 1**: There exist no cache and TLB reuse between different tile traversals.
- **Assumption 2**: W is greater than the cache size and the TLB size and *ITMAX* is large.
- **Assumption 3**: Spatial locality is fully exploited in the innermost loops.

Assumption 1 is reasonable because, with a large *ITMAX*, it is very likely for a tile traversal to overwrite cache lines whose old data could have been reused in the next tile traversal. Many real numerical applications have large data set and require many time steps. Therefore Assumption 2 is reasonable. For regular applications, programmers and existing compilers normally write or transform the programs in such styles that Assumption 3 is naturally satisfied. From Assumption 2, we can derive $NSTEP \ll \gamma$ and $NSTEP \ll (ITMAX\text{-}1)*SLOPE$.

We estimate the cache misses in the original loop nest (Figure 1(a)) as $ITMAX * \frac{W}{C_b}$ and the TLB misses as $ITMAX * \frac{W}{T_b}$.

For the tiled loop nest, we estimate the cache misses as follows. We assume all b_i $(1 \leq i \leq m)$ to be 0 for simplicity, although the discussion can be extended to cover nonzero b_i. Assume that, after odd-even

[2] Within a tile traversal, the exact J-iterations executed in T and $T+1$ overlap partially if *SLOPE* is greater than 0, as in the case of Jacobi (Figure 2(c)).

[3] In this paper, our framework only processes the loops which conform to the program model (Figure 1(a)) and whose data set accessed within each T iteration is T-invariant. Some loop nests from linear algebra codes may not satisfy these conditions, for example, matrix multiplication, LU factorization (preprocessed by creation of single loop iteration), etc. We believe our memory cost model can be extended to such cases.

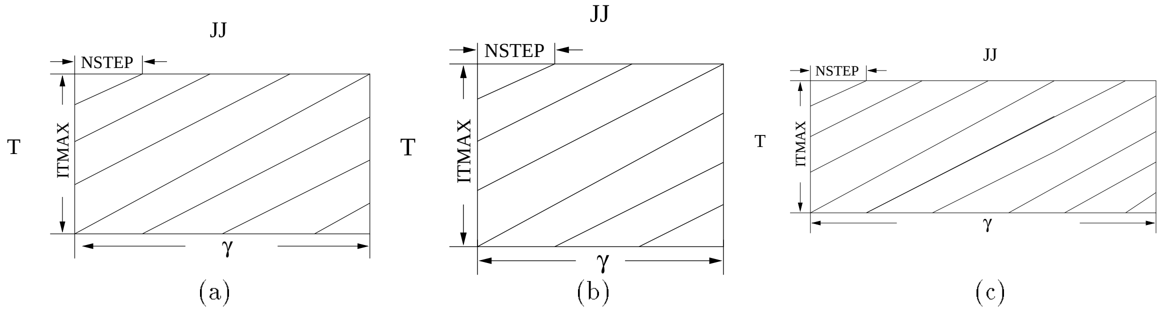


Figure 3: Illustration of Different Scenarios to Calculate the Cache Misses

array duplication as shown in Figure 1(b), the data size of the T loop becomes $\mu W, 1 \leq \mu \leq 2$. The average data size accessed in a single loop tile equals $D = \frac{\mu W}{\gamma} * NSTEP$. The iteration space of the tile-controlling loop JJ spans from γ_1 to $\gamma + \gamma_1 + SLOPE$ *(ITMAX-1)* with the step value *NSTEP* [12], and each JJ value identifies one tile traversal. With Assumptions 1 to 3, we can then estimate the number of cache misses by considering three different cases.

- **Case 1:** $\gamma =$ *(ITMAX-1)* * *SLOPE*.
  This case is illustrated by Figure 3(a). In this case, the tile traversals defined by $JJ \leq \gamma_1 + \gamma - NSTEP$ will not execute to the *ITMAX*th T-iteration. The tile traversal defined by $\gamma_1 + \gamma - NSTEP < JJ \leq \gamma_1 + \gamma$ is the first to reach the *ITMAX*th T-iteration. The tile traversals defined by $JJ > \gamma_1 + \gamma$ will start executing at $T > 1$. During the execution, the tile traversal defined by $JJ = \gamma_1$ will incur cache misses of $\frac{D}{C_b}$. The tile traversal defined by $JJ = \gamma_1 + NSTEP$ will incur cache misses of $\frac{D}{C_b} * 2$, and so on. Hence we have the following:

  - The cache misses in all the tile traversals defined by $JJ \leq \gamma_1 + \gamma - NSTEP$ equal to $\frac{D}{C_b} * (1 + 2 + \ldots + \lfloor \frac{\gamma - NSTEP}{NSTEP} \rfloor)$.
  - Similarly, the cache misses in all the tile traversals defined by $\gamma_1 + \gamma < JJ$ amount to $\frac{D}{C_b} * (1 + 2 + \ldots + \lfloor \frac{\gamma - NSTEP}{NSTEP} \rfloor)$.
  - Similarly, the cache misses in the tile traversal defined by $\gamma_1 + \gamma - NSTEP < JJ \leq \gamma_1 + \gamma$ amount to $\frac{D}{C_b} * \frac{\gamma}{NSTEP}$.

  Adding up the three numbers of the above, the total number of cache misses in the tiled loop nest approximates $\frac{D}{C_b} * (\frac{\gamma}{NSTEP})^2$.

- **Case 2:** $\gamma <$ *(ITMAX-1)*SLOPE*.
  This case is illustrated by Figure 3(b). In this case, one of the tile traversals defined by $JJ > \gamma_1 + \gamma$ first reaches the *ITMAX*th T-iteration. Similar to Case 1, we derive the following:

  - The cache misses in all the tile traversals defined by $JJ \leq \gamma_1 + \gamma$ amount to $\frac{D}{C_b} * (1 + 2 * \ldots + \lfloor \frac{\gamma}{NSTEP} \rfloor)$.
  - The cache misses in all the tile traversals defined by $\gamma_1 + \gamma < JJ \leq$ *(ITMAX-1)*SLOPE* $+ \gamma_1$ amount to $\frac{D}{C_b} * \frac{\gamma}{NSTEP} * \lceil \frac{(ITMAX\text{-}1)*SLOPE - \gamma}{NSTEP} \rceil$.
  - The cache misses in all the tile traversals defined by *(ITMAX-1)*SLOPE* $+ \gamma_1 < JJ$ amount to $\frac{D}{C_b} * (1 + 2 * \ldots + \lfloor \frac{\gamma}{NSTEP} \rfloor)$.

  Adding up the three numbers above, the total number of cache misses in the tiled loop nest approximates $\frac{D}{C_b} * \frac{\gamma}{NSTEP} * \frac{(ITMAX\text{-}1)*SLOPE}{NSTEP}$.

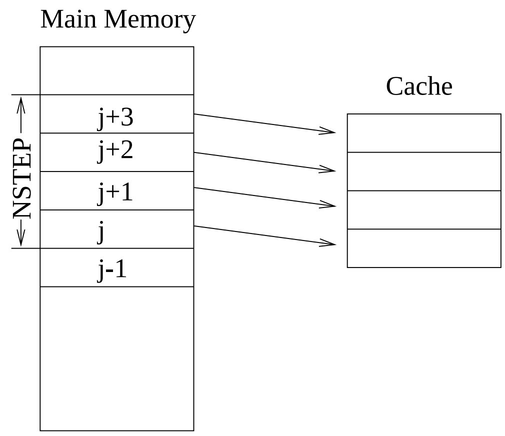


Figure 4: An Illustration to Select a Proper Tile Size

- **Case 3:** $\gamma >$ *(ITMAX-1)*SLOPE.*
  This case is illustrated by Figure 3(c). In this case, one of the tile traversals defined by $JJ \leq \gamma_1 + \gamma - NSTEP$ first reaches the $ITMAX$th T-iteration. Similar to Case 1, we derive the following:
  - The cache misses in all the tile traversals defined by $JJ \leq \gamma_1 + \mathit{(ITMAX\text{-}1)*SLOPE)} - NSTEP$ amount to $\frac{D}{C_b} * (1 + 2 + \ldots + \lfloor \frac{\mathit{(ITMAX\text{-}1)*SLOPE} - NSTEP}{NSTEP} \rfloor)$.
  - The cache misses in all the tile traversals defined by $\gamma_1 + \mathit{(ITMAX\text{-}1)*SLOPE} - NSTEP < JJ \leq \gamma + \gamma_1$ amount to $\frac{D}{C_b} * \frac{NSTEP + \mathit{(ITMAX\text{-}1)*SLOPE}}{NSTEP} * \lceil \frac{\gamma - \mathit{(ITMAX\text{-}1)*SLOPE} + NSTEP}{NSTEP} \rceil$.
  - The cache misses in all the tile traversals defined by $\gamma + \gamma_1 < JJ$ amount to $\frac{D}{C_b} * (1 + 2 + \ldots + \lfloor \frac{\mathit{(ITMAX\text{-}1)*SLOPE\text{-}NSTEP}}{NSTEP} \rfloor)$.

  Adding up the three numbers above, the total number of cache misses in the tiled loop nest approximates $\frac{D}{C_b} * \frac{\mathit{(ITMAX\text{-}1)} * SLOPE}{NSTEP} * \frac{\gamma}{NSTEP}$.

In summary of the above three cases, the total number of cache misses in the tiled loop nest approximates

$$\frac{D}{C_b} * \frac{\mathit{(ITMAX\text{-}1)} * SLOPE}{NSTEP} * \frac{\gamma}{NSTEP} \tag{1}$$

With previously defined $D = \frac{\mu W}{\gamma} * NSTEP$, Equation (1) can be simplied to

$$\frac{W * \mathit{(ITMAX\text{-}1)}}{C_b} * \frac{\mu * SLOPE}{NSTEP} \tag{2}$$

Following similar steps, we estimate the total number of TLB misses in the tiled loop nest to approximately

$$\frac{W * \mathit{(ITMAX\text{-}1)}}{T_b} * \frac{\mu * SLOPE}{NSTEP} \tag{3}$$

In Equations (2) and (3), $ITMAX$, W, C_b and T_b are all T-invariant. In order to reduce the cache misses and the TLB misses, $NSTEP$ should be large and $SLOPE$ should be small, subject to Properties 1 and 2. Property 1 can be preserved by imposing the *working set constraint*, i.e., the amount of data accessed within a single tile should not exceed the cache size and the TLB size. Furthermore, array padding [7] is performed such that the accessed data within a tile are evenly allocated within the cache to minimize the set conflicts. In order to accommodate inaccuracy in array padding and the fact that instructions and data compete for the unified (L2) cache, we follow the industrial practice of utilizing the

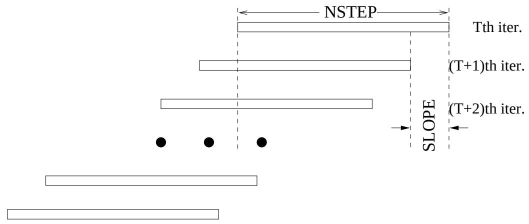


Figure 5: Data Reuse within a Tile Traversal

effective cache size [16], which is smaller than the real cache. (As in our previous work [12], we choose the effective cache size factor $\alpha = 0.3$ in our experiments. Since TLB is fully associative in practice, we choose the effective TLB size to be 0.75, which is considerably greater than the effective cache size factor.)

Figure 4 illustrates how Property 2 is preserved. Suppose that $NSTEP$ equals 4 and $SLOPE$ equals 1 and that the tile executed in a T-loop iteration consists of four consecutive J-loop iterations, denoted by j, $j+1$, $j+2$ and $j+3$ for each J loop. Suppose the data accessed by these four iterations exactly fit in the data cache [4]. In the next T-loop iteration, the tile will contain the iterations $j-1$, j, $j+1$ and $j+2$. Assuming data accessed in consecutive J iterations have consecutive addresses, then a subset of the data accessed in the last tile will be replaced. Since we assume a fully-associative cache, the data accessed previously in j will be replaced. Note that we assume a fully-associative TLB with LRU replacement policy. If the data accessed in a tile which contains iteration j through $j+3$ exactly fit in the TLB in a particular T-loop iteration, then the TLB entry accessed previously in j will be replaced in the next tile when T increases by 1.

We now determine the tile size, $NSTEP$, as follows. Let κ represent the maximum number of J iterations whose data can fit in both the cache and the TLB under the working set constraint. The tile size is chosen to be $NSTEP = \kappa - SLOPE$. This ensures that, within the same tile traversal, data shared by two consecutive tiles will not be replaced before used by the latter tile [5].

Equations (2) and (3) give the number of estimated cache and TLB misses. For a given loop nest and a given machine architecture, the number of cache and TLB misses is completely determined by $\frac{\mu * SLOPE}{NSTEP}$. In this paper, we call its reciprocal, $\lceil \frac{NSTEP}{\mu * SLOPE} \rceil$, the *exploited reuse factor*. Therefore, the greater exploited reuse factor, the smaller the number of the cache and TLB misses.

To give an intuitive understanding of exploited reuse factor. Figure 5 shows the overlapping pattern of one J loop, say J_i, within a tile traversal. Among all $T+i$ iterations, the degree of overlap between the J-iterations executed in T and those in $T+i$ decreases as i increases. As i becomes greater than $\lceil \frac{NSTEP}{SLOPE} \rceil$, no more overlap exists between T and $T+i$. Since array duplication increases memory requirement, we can regard μ as a penalty factor. Our objective is to maximize $\lceil \frac{NSTEP}{\mu * SLOPE} \rceil$ such that the reuse between T and $T + \lceil \frac{NSTEP}{SLOPE} \rceil$ can be fully realized by satisfying Properties 1 and 2.

4 Identifying Compatible Loops

In this section, we present an algorithm to determine the compatible loops associated with the given loop nest T. In order to do that, we adopt a data structure called the *orientation graph*, which is previously used for parallel loop scheduling and data allocation [10]. An *orientation graph OG* is a graph which associates loops with array dimensions. For Jacobi code shown in Figure 2(a), the orientation graph is shown in Figure 6 where A_i and L_i represent the data in the first ($i = 1$) or second ($i = 2$) dimension

[4] For a cache with limited set associativity, this may not be true because, within a tile, the references from different arrays may map to the same cache locations, which may cause one portion of the cache to be underutilized and the other portion oversubscribed. We expect that data transformation techniques such as padding [7, 11] can relieve this problem in reality.

[5] For directly-mapped caches, if the TLB is fully-utilized, the tile size should also be $\kappa - SLOPE$, otherwise, it is simply κ. For all other set-associative caches, the tile size should be $\kappa - SLOPE$.

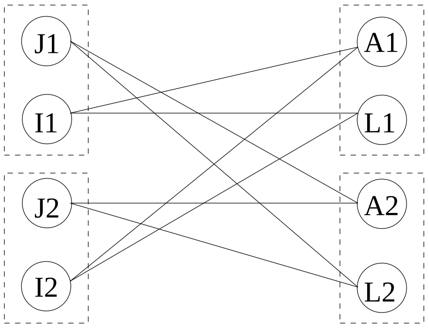


Figure 6: Orientation Graph for Jacobi Kernel

Input: A loop nest LP and its J-loop distance subgraph G_l.
Output: the exploited reuse factor r and the profitability p
Procedure:
Perform *circular loop skewing* on LP.
Compute the minimum slope $SLOPE$ by only considering flow dependences.
Perform array duplication.
Perform forward substitution.
Recompute $NSTEP$ and perform profitability test.
Return the profitability p and if profitable, the exploited reuse factor $r = \lceil \frac{NSTEP}{\mu * SLOPE} \rceil$.

(a)

Input: A loop nest LP and its J-loop distance subgraph G_l.
Output: the exploited factor reuse r and the profitability p
Procedure:
Perform *circular loop skewing* on LP.
Compute the minimum slope $SLOPE$ by considering flow, anti- and output dependences.
Recompute $NSTEP$ and perform profitability test.
Return the profitability p and if profitable, the exploited reuse factor $r = \lceil \frac{NSTEP}{SLOPE} \rceil$.

(b)

Figure 7: Tiling With and Without Duplication

respectively. The loop nests within each dashed-line box are perfectly-nested. The arrays within each dashed-line box have their data in the same dimension accessed by some loops.

Given a T loop conforming to our program model, suppose each J_i loop contains s_i inner loops, $I_{i,1}$, $I_{i,2}$, ..., I_{i,s_i}, $s_i \geq 0$, which are perfectly nested within J_i from the outmost to the innermost. Denote J_i by $I_{i,0}$. Our algorithm generates all the compatible sets, and it records the arrays which are associated with each compatible set. The algorithm works as follows:

1. For all the loops which have accesses to the first dimension of some arrays, select one $I_{i,k}, 0 \leq k \leq s_i$ for each $1 \leq i \leq m$ to form the compatible set (I_{1,k_1}, I_{2,k_2}, ..., I_{m,k_m}, ROW), and regard those arrays whose first dimensions are accessed by $I_{i,k_i}, 1 \leq i \leq m$, as the arrays associated with this compatible set. We collect all such row-compatible sets. If the loops $I_{i,k}, 0 \leq k \leq s_i$ do not access the first dimension for any array, we fail to generate any row-compatible sets.

2. Find all the column compatible sets following steps similar to the above.

Column-compatible sets are preferred to row-compatible sets, and hence the former are inspected first. In Jacobi (Figure 2(a)), we find two compatible sets, (I_1, I_2, ROW) and (J_1, J_2, $COLUMN$).

5 Tiling With and Without Duplication

In our framework, the compiler tries tiling both with and without odd-even array duplication, and picks the one with a greater exploited reuse factor. In the following we discuss how to perform tiling with and without array duplication respectively.

5.1 Tiling with Duplication

As in our previous work [12], Figure 1(b) is the result after array duplication, but before tiled-code generation, and Figure 1(c) is the tiled code. Note that the computation of the tile size $NSTEP$ is now based on our new memory cost model in Section 3, not simply based on the working-set constraint as in [12]. Figure 7(a) sketches the procedure to perform tiling with duplication, where *circular loop skewing* refers to a technique discussed in [12] to shorten long backward dependence distance.

In Figure 7(a), the profitability test is as follows.

1. Condition $NSTEP > SLOPE$ must be satisfied.
2. Let $NSTEP1 = max\{offset[v] + |d_J| - offset[u] \mid u, v$ are nodes in $G_l, < u, v >$ is any anti- or output dependence edge with $d_T = 0.\}$, $NSTEP2 = max\{offset[v] + |d_J| - offset[u] - SLOPE \mid u, v$ are nodes in $G_l, < u, v >$ is any anti- or output dependence edge with $d_T = 1.\}$.
   Condition $NSTEP > max\ (NSTEP1,\ NSTEP2)$ must be satisfied.

Condition 1 makes $NSTEP$ large enough to allow potential temporal reuse across T iterations. Condition 2 prevents any anti- and output dependence from crossing backward more than one tile boundary. If the final loop nest passes profitability test, profitability $p = TRUE$ and the exploited reuse factor $r = \lceil \frac{NSTEP}{\mu * SLOPE} \rceil$ are returned. Otherwise, profitability $p = FALSE$ is returned. Figure 8 is the transformed Jacobi code if transformation with duplication is applied solely.

5.2 Tiling without Duplication

Alternative to odd-even duplication, we can conservatively select $SLOPE$ and *offset*s such that no backward dependences (flow, anti- and output) exist from a later tile to an earlier tile. Figure 7(b) shows the basic procedure for tiling without duplication. The final code form of tiling without duplication is shown in Figure 1(d). The main difference from tiling with odd-even duplication is that, when we compute $SLOPE$ and *offset*s [12], we examine not only the flow dependences but also the anti- and output dependences. If we tile the loops without duplication, then we do not need to perform forward substitution. Moreover, the profitability test in Section 5.1 no longer needs to check Condition 2.

For example, if Jacobi kernel is tiled with duplication, $SLOPE$ equals to 1, and both the offset values for both J_1 and J_2 equal 0. If it is tiled without duplication, however, $SLOPE$ equals 2, the offset value of J_2 equals 0 and that of J_1 equals 1. Figure 9 shows the final code for Jacobi tiled without duplication.

6 An Integrated Framework

In this section, we present our integrated compiler framework to tile imperfectly-nested loops, which is followed by a running example.

Figure 10 presents the sketch of the framework. In this framework, the compiler first identifies all loops each of which may be labeled as the T loop. We say these are the *T candidates*. The compiler then computes all the dependence information associated with each T candidate [12].

If a loop nest contains more than one T candidate, the compiler orders the T candidates to form a priority queue, Q, based on the *reuse factor* which is defined as the maximum number of times any element is referenced by a loop repeatedly at the given loop level [15, 17]. In case that two T candidates have the same reuse factor, if one is nested within the other, the inner one has a higher priority. Otherwise, the order can be arbitrary. The compiler extracts one T candidate, say T_1, from Q. If T_1 can be tiled successfully, the tiling process terminates for the given loop nest. Otherwise, the compiler picks the next T candidate, until Q is empty.

For each T candidate, T_1, the compiler first computes all the compatible sets based on the algorithm presented in Section 4. If the number of compatible sets exceeds the maximum number of perfectly-nested loop levels with T_1, then tiling T_1 is unlikely to be profitable, and the compiler moves to the next T candidate. Otherwise, the compiler examines all the compatible sets with T_1 as the T loop, starting from column-compatible ones, as follows.

For each compatible set, *cset*, the compiler first tries to transform the loop nest according to *cset* by loop permutation and array transpose. If such transformations are impossible, the compiler continues to the next compatible set. Otherwise, the J-loop distance subgraph is constructed [12]. The compiler

```
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: Final Code for Jacobi with Duplication before Loop Fusion

then tries to tile with and without odd-even duplication. If both tiling schemes are profitable, then the one with more exploited reuse is favored. If neither is profitable, the compiler continues to the next compatible set.

6.1 A Running Example

We now go through Jacobi (Figure 2(a)) as an example, assuming that N equals 512, and that the data element is of double precision and $ITMAX$ equals 100. Suppose that the cache size is 64K and the cache block size 16, both in terms of data element size. Suppose that TLB has 96 entries where each entry represents a virtual page of 1K data elements. Thus, the TLB block size is 1K, and the TLB size is 96K. As stated in Section 3, we set the effective cache size factor to 0.3 and the effective TLB size factor to 0.75.

- We have $TPool = \{T, J_1, J_2\}$. The reuse factor at the T level is $ITMAX$ and is 3 at the J_1 and J_2 levels. Assuming $ITMAX$ is greater than 3, the possible orders of these three T candidates in the priority queue are (T, J_1, J_2) or (T, J_2, J_1). The compiler selects T as the first T-candidate.
- We have $comSets = \{(J_1, J_2, COLUMN), (I_1, I_2, ROW)\}$. $s_1 = s_2 = 1$. The compiler favors (J_1, J_2, $COLUMN$) and it performs the following the steps.
  - No loop permutation or array transpose is needed since the compatible set is column associated and J_i's are immediately nested in T.
  - Tile with duplication: (Circular loop skewing has no effect on Jacobi.) We have $SLOPE = 1$. Array duplication makes a copy of A, say C, but forward substitution eliminates L. Therefore, $\mu = 1$. Using the memory cost model, we have $NSTEP = 61$. Finally, the compiler determines tiling to be profitable and it computes the exploited reuse factor $r = 61$.

```
DO JJ = 2, N + 2 * ITMAX - 3, NSTEP
  DO T = max((JJ-N+3)/2, 1), min((JJ+NSTEP)/2, ITMAX)
    DO J1 = max(3, JJ - 2 * T + 3),
          min(N - 1, JJ - 2 * T + NSTEP + 2)
      IF (J1.EQ.3) THEN
        DO K1 = 2, 2
          DO I1 = 2, N - 1
            L(I1, K1) = (A(I1 + 11, K1) + A(I1 - 1, K1) + A(I1, K1 + 1) + A(I1, K1 - 1))/4
          END DO
        END DO
      END IF
      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 - 2 * T + 2),
          min(N - 1, JJ - 2 * T + NSTEP + 1)
      DO I2 = 2, N - 1
        A(I2, J2) = L(I2, J2)
      END DO
    END DO
  END DO
END DO
```

Figure 9: Final Code for Jacobi without Duplication before Loop Fusion

Input: a loop nest.
Output: a transformed loop nest if the algorithm succeeds or the restored original loop nest if the algorithm aborts.
Procedure:

```
Compute a set of T candidates, say TPool, and   dependence information associated with each T candidate.
Order the T candidates in TPool to form a priority queue Q.
do
  if (Q is empty) aborts.
  Extract a T loop from Q, say T1.
  Compute the compatible sets for T1, say comSets.
  if (|comSets| > max {s_i + 1|1 ≤ i ≤ m}) continue to the next do iteration.
    [ see Section 2.2 for explanation of s_i. ]
  for each compatible set cset do
    Transform the loop nest according to cset.
    if (this transformation cannot be realized) continue to the next compatible set.
    Construct J-loop distance subgraph.
    Transform the loop nest with duplication with the exploited reuse factor r1 and the profitability p1 returned.
    Transform the loop nest without duplication with the exploited reuse factor r2 and the profitability p2 returned.
    if (p1) perform real transformation with duplication if p2 is FALSE or r1 > r2.
    if (p2) perform real transformation without duplication if p1 is FALSE or r2 ≥ r1.
    if (p1 or p2) return from the procedure.
  end for
end do
abort.
```

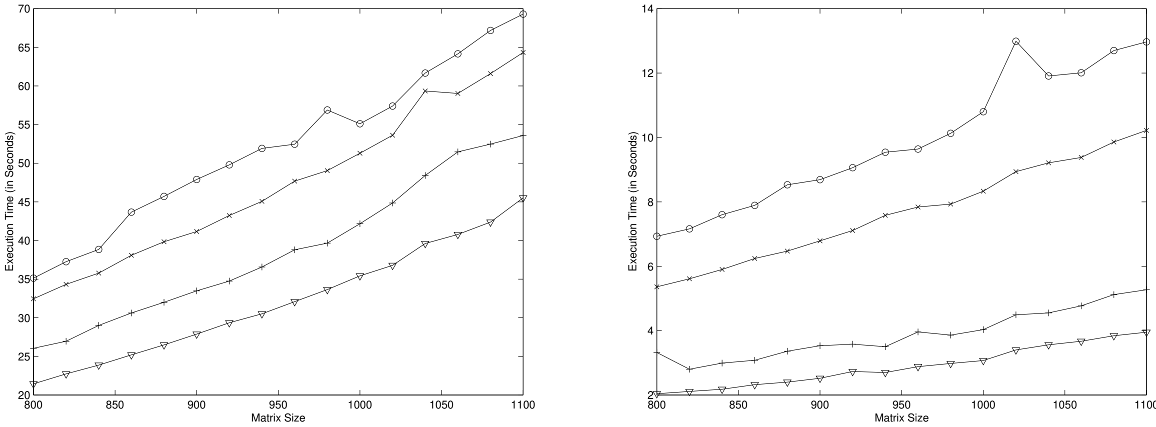
Figure 10: A Sketch of Compiler Framework to Tile Imperfectly-Nested Loops

  - Tile without duplication: We have *SLOPE = 2*. Tiling is again profitable and the exploited reuse factor is $r = 31$.
  - The compiler chooses tiling with duplication since it has a greater exploited reuse factor.

- The compiler does not process the compatible set (I_1, I_2, *ROW*) since it already finds a profitable tiling scheme.

7 Experimental Evaluation

We are implementing the new framework presented above in the Panorama compiler [4]. To pre-evaluate their effectiveness, we have manually applied this framework to two well-known numerical kernels, Jacobi and Livermore Loop LL18, and two SPEC95 benchmarks `tomcatv` and `swim`, which run on two processors. One is a MIPS R5K processor within an SGI workstation, and the other is a single MIPS R10K processor within an SGI Origin 2000 multiprocessor. The R5K processor has a 32KB 2-way L1 data cache, a 512KB 2-way unified L2 cache and a 48-entry fully-associative unified TLB. Each TLB entry represents an 8KB



(a) On the R5K (b) On the R10K

Figure 11: Execution Time (in Seconds) of Jacobi on the R5K and R10K with Various Matrix Size

page. The R10K processor has a 32KB 2-way L1 data cache, a 4MB 2-way unified L2 cache and a 64-entry fully-associative unified TLB. Each TLB entry also represents an 8KB page [6]. Both machines have a random replacement policy. 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. We turn on the "-O3" switch for the original code, but turn off the tiling switch in the compiler for all the other schemes. The "-O3" option enables a number of loop transformations including interchange, fusion, fission and tiling. For peel-and-fusion, we also apply padding to reduce cache set conflicts and choose the tile size such that the working set for any array within a single tile can fit in one partition [7], where both the real cache size and effective cache size are used in experiments and the better results are taken as the final one. We use both the real cache size and the effective cache size and take the better result as the final one. We apply a speculation scheme [12] to overcome the convergence test in Jacobi and in `tomcatv`. The basic idea of speculation is to partition the whole T iteration space into several chunks, so tiling can be applied to each individual chunk. Checkpointing is performed before executing a new chunk. The convergence test is performed after the execution of a chunk. In case of convergence test failure, the latest checkpoint is restored and the execution is completed by the untiled version.

The Jacobi Kernel

We fix ITMAX to 100 and vary the input matrix size arbitrarily from 800 to 1100 incremented by 20. Figure 11(a) and (b) show the performance result for different versions of Jacobi code on the R5K and R10K respectively.

NSTEP is computed based on the scheme presented in Section 3. The compiler chooses tiling with duplication since it yields a greater exploited reuse factor. The data for non-duplication tiling is presented for comparison.

The final code of Jacobi does not increase the memory usage because forward substitution eliminates the usage of L as shown in Figure 9. Figure 11 shows that tiling with duplication performs better than the other three schemes. On the R10K, it improves the performance by 227% to 282% over the original code and by 156% to 181% over peel-and-fusion. On the R5K, it improves the performance by 52% to 73% over the original code and by 41% to 51% over peel-and-fusion.

[6] The R10K machine used here has a larger L2 cache and a faster clock than the one used in our previous work [12]. We achieve a higher speedup with the new techniques in this paper.

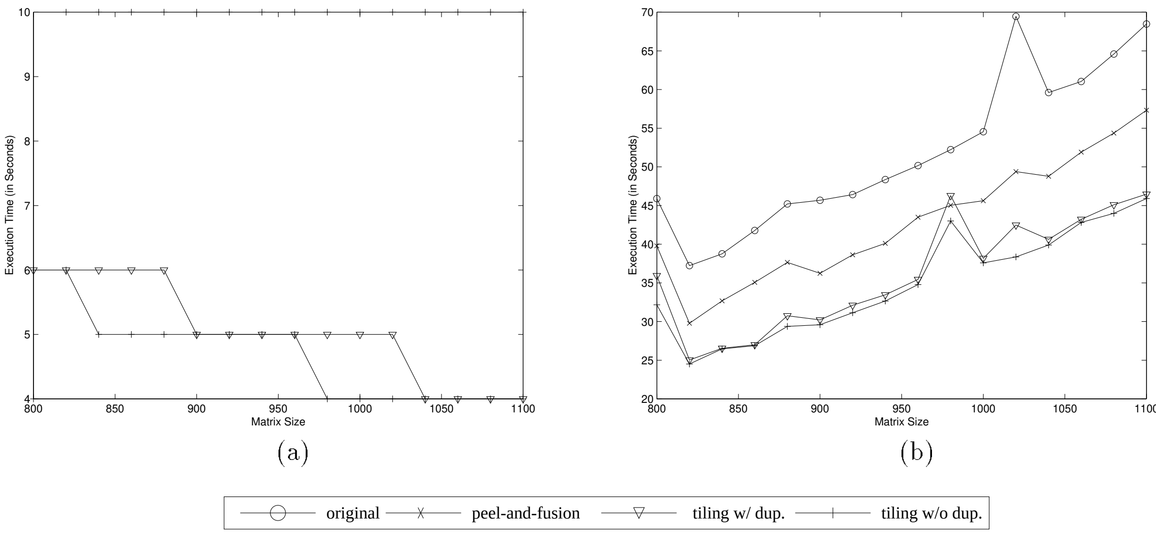


Figure 12: The Exploited Reuse Factor and Execution Time (in Seconds) of LL18 on the R10K with Various Matrix Size

The LL18 Kernel

We fix ITMAX to 100 and choose the same input matrix sizes as for Jacobi. With two arrays, *ZR* and *ZZ*, duplicated, *SLOPE* is computed as 2. The memory requirement increases by 22%. Without duplication, *SLOPE* equals 3. Figure 12(a) shows the exploited reuse factor for the schemes with and without duplication on the R10K. The compiler favors tiling with duplication when N equals to 840, 860, 880, 980, 1000 and 1020. The compiler favors tiling without duplication for all the other cases where the exploited reuse factors are equal.

Figure 12(b) shows the performance result for various versions of LL18 on the R10K. For the schemes with and without duplication, the difference between the exploited reuse factors is at most 1, predicting a similar performance. The execution time of the code transformed with the two schemes is indeed very close except for $N = 800, 980$ and 1020. Although our framework does not choose the best solution for all cases, the result of the transformed code still outperforms the original code by 13% to 64%, and it outperforms peel-and-fusion by 16% to 30% except $N = 980$, where it degrades the performance by 3% compared with peel-and-fusion. With such a large data set and our chosen effective cache factor, our framework cannot profitably perform tiling on the R5K. With a smaller size of data set, however, we get results similar to what are shown in Figure 12.

`tomcatv`

Two arrays, X and Y, among the 7 N-by-N arrays in `tomcatv`, are duplicated by the duplication algorithm, increasing the memory usage by 29%. On the R10K, *NSTEP* equals 33 if we tile with duplication and it equals 41 without duplication. On the R5K, it equals 3 with or without duplication. Since *SLOPE = 1* with duplication and *SLOPE = 2* without duplication, the compiler chooses tiling with duplication for both machines.

Using the reference input data, `tomcatv` always runs to the maximum time step, i.e., 750. On the R10K, the original program completes in 116 seconds. The transformed program with duplication completes in 81 seconds and the one without duplication completes in 86 seconds, a speedup of 1.43 and 1.34 respectively. On the R5K, the original program completes in 732 seconds. The transformed program with duplication completes in 451 seconds and 528 seconds with and without duplication respectively, giving a speedup of 1.62 and 1.39. We also run `tomcatv` with peel-and-fusion, where the transformed

Table 2: Execution Time(in Seconds) of `swim`

Test Program	R10K		R5K	
	Exec. Time	Speedup	Exec. Time	Speedup
Orig. Prog.	92.66	1.00	625	1.00
Trans. Prog.	58.55	1.58	573	1.09

Table 3: Secondary Cache Utilization of Jacobi, LL18, `tomcatv` and `swim`
(LS and SM are counters in millions)

Test Program	Original			Peel-and-Fusion			Duplication			Non-duplication		
	LS	SM	MR	LS	SM	MR	LS	SM	MR	LS	SM	MR
Jacobi (N = 1000)	553.5	24.9	0.045	648.5	12.5	0.019	451.1	0.45	0.0010	671.7	0.53	0.0008
LL18 (N = 1000)	2767.4	99.7	0.036	2762.7	56.3	0.020	3400.8	12.8	0.0038	3403.2	15.1	0.0044
tomcatv	10809.2	177.1	0.016	10052.3	142.4	0.014	8335.2	51.2	0.0061	8151.4	73.1	0.0090
swim	11701.2	167.1	0.014	-	-	-	9280.4	5.0	0.0005	9280.4	5.0	0.0005

code takes 132 and 677 seconds on the R10K and the R5K and achieves a speedup of 0.87 and 1.08 respectively over the original program.

`swim`

In `swim`, *SLOPE* equals 2 and *NSTEP* equals 45 and 3 for the R10K and the R5K respectively. No arrays need to be duplicated to decrease *SLOPE*. Table 2 shows the performance results, where "Orig. Prog." stands for the original program, and "Trans. Prog." represents the transformed program. On the R5K the speedup by tiling is lower than on the R10K. We suspect that, for the R5K, the small *NSTEP* value, hence the small overlap of iteration points between consecutive tiles in the same tile traversal, has made our tiling results more sensitive to the cache set conflicts. Peel-and-fusion does not apply directly to `swim` due to long backward dependence distances.

7.1 Cache Miss Measurement

On the R10K, in addition to execution time, we also measure the secondary cache misses using *perfex* library based on R10K performance counters. Table 3 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. We show the results for the Jacobi and LL18 with N arbitrarily chosen as 1000. We also show results for `tomcatv` and `swim` using the reference data. In Jacobi, even with checkpointing, the number of dynamic load-store instructions are reduced due to forward substitution. In all four test programs the secondary cache miss rate is dramatically reduced by our tiling scheme. The gain in temporal locality across T iterations offsets the loss due to more memory requirements after array duplication.

8 Related Work

Kodukula *et al.* propose *data shackling* [5], which blocks the arrays based on data flow analysis and then forms new loop nests to compute 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.

Manjikian *et al.* present peel-and-fusion and apply padding to eliminate cache set conflict within a tile [7]. 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 offset computation 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.

Bacon *et al.* develop a padding algorithm for selecting efficient padding amount, which takes into account both cache and TLB in a single framework [1]. Their method mainly applies to the innermost loop. More recently, Rivera and Tseng present a set of padding heuristics which are used to perform inter- and intra-array padding [11]. Padding and tiling are generally viewed as two complementary techniques.

In our framework, any repeated reference to a memory location within the same tile will be a cache hit and a TLB hit, so the working set for that tile will determine the number of cache misses and TLB misses. Ferrante *et al.* provide closed-form formulae that bound the number of array accesses and the number of cache lines accessed within a loop nest, thus providing an estimate of the number of cache misses in a loop nest [2]. Their method applies to perfectly-nested loops only. Temam *et al.* presents an algorithm to estimate the number of cache misses, especially interference misses [14], for a perfectly-nested loop nest. Ghost *et al.* develop a cache miss equation (CME) to count the number of cache misses in order to guide optimization [3] for a perfectly-nested loop. Our work in this paper covers more complex loop nests, including imperfectly-nested loops and skewed tiles.

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 [9], although they provide no formal algorithms to select tile sizes.

Kodukula and Pingali propose a matrix-based framework to represent transformations of imperfectly-nested loops [6] including permutation, reversal, skewing, scaling, alignment, distribution and jamming, but not including tiling.

McCalpin and Wonnacott introduce the notion of *machine balance* vs. *compute balance*, and they develop a scheme called *time skewing* which adopts a value-based flow analysis to optimize for memory locality [8]. Their method first performs full array expansion and forward substitution, and it then recompresses the expanded array while preserving data dependences. Their current method handles a subset of the imperfectly-nested loops represented by our program model.

9 Conclusion and Future Work

In this paper, we have extended our previous work [12] on tiling imperfectly-nested loops. We present a framework to systematically select and tile the given loop candidates. We are implementing our new techniques in a Fortran source-to-source compiler, Panorama [4]. Preliminary experimental results show the transformed programs run faster by 9% to 282% than those optimized by the native compiler only.

Our work may be further improved in a number of ways. The scheme for ordering the T candidates may be improved further. We also need to better understand the interaction between the TLB and the multiple levels of cache memories, and the competition between instructions and data for unified L2 cache, among other issues, which can affect the tile-size selection.

References

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

[2] J. Ferrante, V. Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness. In *Proc. of 4th Int'l Workshop on Languages, Compilers for Parallel Computing*, August 1991.

[3] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. In *Prof. of 8th ACM Conf. on ASPLOS*, San Jose, California, October 1998.

[4] J. Gu, Z. Li, and G. Lee. Experience with efficient array data flow analysis for array privatization. In *6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 1997.

[5] Induprakas Kodukula, Nawaaz Ahmed, and Keshav Pingali. Data-centric multi-level blocking. In *ACM SIGPLAN conference on PLDI*, June 1997.

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

[7] Naraig Manjikian and Tarek Abdelrahman. Fusion of loops for parallelism and locality. In *IEEE Transactions on Parallel and Distributed Systems*, volume 8, Feb 1997.

[8] 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.*

[9] Nicholas Mitchell, Karin Högstedt, Larry Carter, and Jeanne Ferrante. Quantifying the multi-level nature of tiling interactions. In *International Journal of Parallel Programming*, 1998.

[10] Trung Nguyen and Zhiyuan Li. Interprocedural analysis for loop scheduling and data allocation. In *Parallel Computing*, volume 24, pages 477–504, 1998.

[11] 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.

[12] Yonghong Song and Zhiyuan Li. New tiling techniques to improve cache temporal locality. In *ACM SIGPLAN conference on PLDI*, pages 215–228, May 1999.

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

[14] O. Temam, C. Fricker, and W. Jalby. Cache interference phenomena. In *Proc. of ACM SIGMETRICS Conf. on Measurement & Modeling of Computer Systems*, May 1994.

[15] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *ACM SIGPLAN conference on PLDI*, June 1991.

[16] 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.

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