MULTILEVEL TILING FOR NON-RECTANGULAR ITERATION SPACES

Marta Jiménez

Departamento de Arquitectura de Computadores
Universitat Politècnica de Catalunya
Barcelona (Spain). March, 1999

A THESIS SUBMITTED IN FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE
Doctor por la Universitat Politècnica de Catalunya

4

SIMULTANEOUS MULTILEVEL TILING

Summary

This chapter presents a new cost-effective algorithm to compute exact loop bounds when Multilevel Tiling is applied to a loop nest having affine functions as bounds. Traditionally, exact loop bounds computation has not been performed because its complexity is doubly exponential on the number of loops in the multilevel tiled code and, therefore, for certain classes of loops, can be extremely time consuming. Although computation of exact loop bounds is not very important when tiling only for cache levels, it is critical when tiling includes the register level. As shown in Chapter 3, both the cost of tiling for the register level and the amount of code generated depend on the number of loop bounds in the multilevel tiled code. This chapter presents a new Multilevel Tiling technique that computes exact loop bounds whose complexity is much lower than the complexity of conventional techniques. To achieve this lower complexity, our technique deals simultaneously with all levels to be tiled, rather than applying tiling level by level as is usually done. This chapter will compare our implementation against conventional techniques in terms of complexity and loop bounds generated and present some experimental results.

4.1 INTRODUCTION

Loop Tiling is a transformation that a compiler can use not only to achieve data locality in different levels of the memory hierarchy [27][83][97][122], but also to exploit parallelism [69][84][123]. With today's architectures having complex memory hierarchies and multiple processors, it is quite common that the compiler has to perform tiling at four or more levels (parallelism, L2-cache, L1-cache and registers) in order to achieve high performance.

Multilevel tiling consists of dividing a tile of a higher level into smaller subtiles, where each level of tiles exploits one level of the memory hierarchy or one level of parallelism. As shown in Chapter 2, conventional tiling techniques implement one level of tiling using repeatedly the strip-mining and the loop permutation transformations [129]. Then, to implement multilevel tiling several researchers propose applying tiling level by level [28][121], going from the outermost (parallelism) to the innermost level (registers).

Previous research in multilevel tiling code generation can be divided in two main groups: techniques that compute the resulting tiled loop nest with exact loop bounds [7][77] and techniques that do not compute exact loop bounds [122]. We say that a loop nest has *exact* bounds if it never executes an empty iteration (recall the example of Fig. 1.5 on page 29). Clearly, a loop nest with exact bounds is more efficient because the loop will not waste time in empty iterations. However, to date, the drawback of generating exact loop bounds was that all techniques known were extremely expensive and, thus, difficult to integrate in a production compiler.

Another problem related to the generation of multilevel tiled loop nests is the generation of *redundant* bounds [5][16]. We say that a loop bound is *redundant* if it can be removed from the loop and the resulting loop nest executes exactly the same iterations as the original loop nest. Current techniques able to eliminate redundant bounds are also very expensive in terms of compilation time and, as above, have not been included in production compilers.

Solving the problems of exact and redundant bounds would be very beneficial for two main reasons: first, as just mentioned, it avoids increasing a program's execution time. If the compiler does not compute exact bounds and generates redundant bounds, a fraction of a program's execution time is wasted in evaluating useless bounds (redundant bounds or bounds of loops that will end up in empty iterations). This fraction of time is insignificant if tiling is applied to rectangular iteration spaces or for one or two levels of the memory hierarchy. However, it can be very important if tiling is applied to non-rectangular iteration spaces and for several levels of the memory hierarchy.

Second, and most important, computing *exact* bounds and avoiding the generation of redundant bounds is critical when multilevel tiling includes the register level. The reason is that, as shown in Chapter 3, the number of times that Index Set Splitting is applied and the amount of code generated both depend polynomially on the number of bounds of the loops that have to be fully unrolled (the

innermost loops after tiling). If the number of generated loop nests increases excessively, the compiler might waste a lot of time performing the instruction scheduling and the register allocation of loop nests that will be never executed. Thus, when multilevel tiling includes the register level, it is convenient to compute exact bounds and to eliminate redundant bounds, at least in the innermost loops.

Traditionally, exact loop bounds computation has not been performed because its complexity is doubly exponential on the number of loops in the multilevel tiled code and, therefore, for certain classes of loop nests, can be extremely time consuming. Of course, simple loop nests that define rectangular iteration spaces, incur in the best-case complexity and, for this type of loop nests, the cost of computing exact bounds is linear on the number of loops in the multilevel tiled code. However, this is not the case for complex loop nests defining non-rectangular iteration spaces. These complex loop nests are commonly found in linear algebra programs or can arise as a result of applying transformations such as loop skewing[1]. Moreover, conventional multilevel tiling implementations generate many redundant bounds in the innermost loops and eliminating these redundant bounds is also a very time consuming job that can increase a program's compile time significantly [5][16].

In this chapter we present a new implementation of multilevel tiling that computes exact loop bounds at a much lower complexity than traditional techniques. Moreover, our implementation generates less redundant bounds in the multilevel tiled code and allows removing the remaining redundant bounds in the innermost loops at a much lower cost than traditional implementations. Using our algorithm, tiling for the register level becomes viable even in the face of complex loop nests and/or when tiling for many levels. The main idea behind our algorithm is that we deal with all levels to be tiled simultaneously, instead of applying tiling level by level as traditional implementations do. We evaluate analytically the complexity of our implementation and show that it is proportional to the complexity of performing a loop permutation in the original loop nest, while conventional techniques have much larger complexities. We then compare our implementation against traditional techniques for typical linear algebra codes having very simple affine functions as bounds and show that our method is between 1.5 and 2.8 times faster. Moreover, for loop nests having not so simple bounds[2] (but still bounds commonly found in linear algebra programs), the speedups achieved can be as high as 2300. We also compare both implementations in terms of redundant bounds generated and cost of eliminating these redundant bounds. We show that eliminating redundant bounds in a multilevel tiled code generated with our proposal is between 2.2 and 11 times faster than in a code generated with conventional techniques.

The rest of this chapter is organized as follows. In Section 4.2 we briefly review how conventional techniques implement multilevel tiling and evaluate their complexity. In Section 4.3 we show how multilevel tiling can be performed dealing with all levels simultaneously. In Section 4.4 we give our

1.As shown in Chapter 2, loop skewing is sometimes necessary to allow loop tiling to be applied.
2.Bounds that are affine functions of multiple outer loop index variables

efficient implementation of multilevel tiling and in Section 4.5 we evaluate its complexity. In Section 4.6 we show how our technique avoids the generation of some special redundant bounds and reduces the cost of eliminating the remaining redundant bounds. In Section 4.7 we compare our implementation against conventional techniques in terms of complexity, number of redundant bounds generated and cost of eliminating redundant bounds. In Section 4.8 we present the previous work related to multilevel tiling and, in Section 4.9, we summarize this chapter.

Finally, we want to note that, in this chapter, multilevel tiling implementation refers only to the compilation phase of updating the transformed loop nest, that is, it refers to computing the loop bounds in the final tiled code.

4.2 CONVENTIONAL TILING IMPLEMENTATION

In this section we briefly review how a conventional technique implements multilevel tiling and we evaluate its complexity.

From now on, we assume that the loop bounds in the original code are max or min functions of affine functions of the surrounding loops iteration variables. We will refer to each affine function as a *simple bound*. We also assume that the loop nest to be tiled is fully permutable and perfectly nested. Extensions to handle non-perfectly nested loops were explained in Chapter 2. Another important assumption is that the bounds in the original code are exact. If they were not exact, it would be necessary to apply the Fourier-Motzkin Elimination algorithm[3] to obtain the exact bounds [16][77].

4.2.1 Implementation

As shown in Chapter 2, conventional tiling techniques implement one level of tiling using two well-known transformations: strip-mining and loop permutation [123][129]. Strip-mining is used to partition one dimension of the iteration space into strips and the loop permutation is used to establish the order in which the iterations inside the tiles are traversed. To perform one level of tiling, it is usually necessary to partition the iteration space in more than one dimension (multi-dimensional tiling). Conventional techniques apply strip-mining and loop permutation repeatedly, as many times as dimensions have to be partitioned[4].

Figure 4.1 reviews how conventional tiling implementations work. We are tiling two dimensions of the 3-dimensional iteration space (two-dimensional tiling) where B_{JJ} and B_{II} are the tile sizes in each dimension. Conventional implementations perform both transformations (strip-mining and loop permutation) as many times as dimensions have to be partitioned. Strip-mining decomposes a loop

3.By applying the Fourier-Motzkin Elimination algorithm to the original code, independently of whether a loop permutation is performed or not, the bounds of the Minimum Convex Space described by the loop structure are obtained.
4.A previous loop permutation must be performed if the loop order of the original loop nest is not such that the outermost loop is the loop to be strip-mined first.

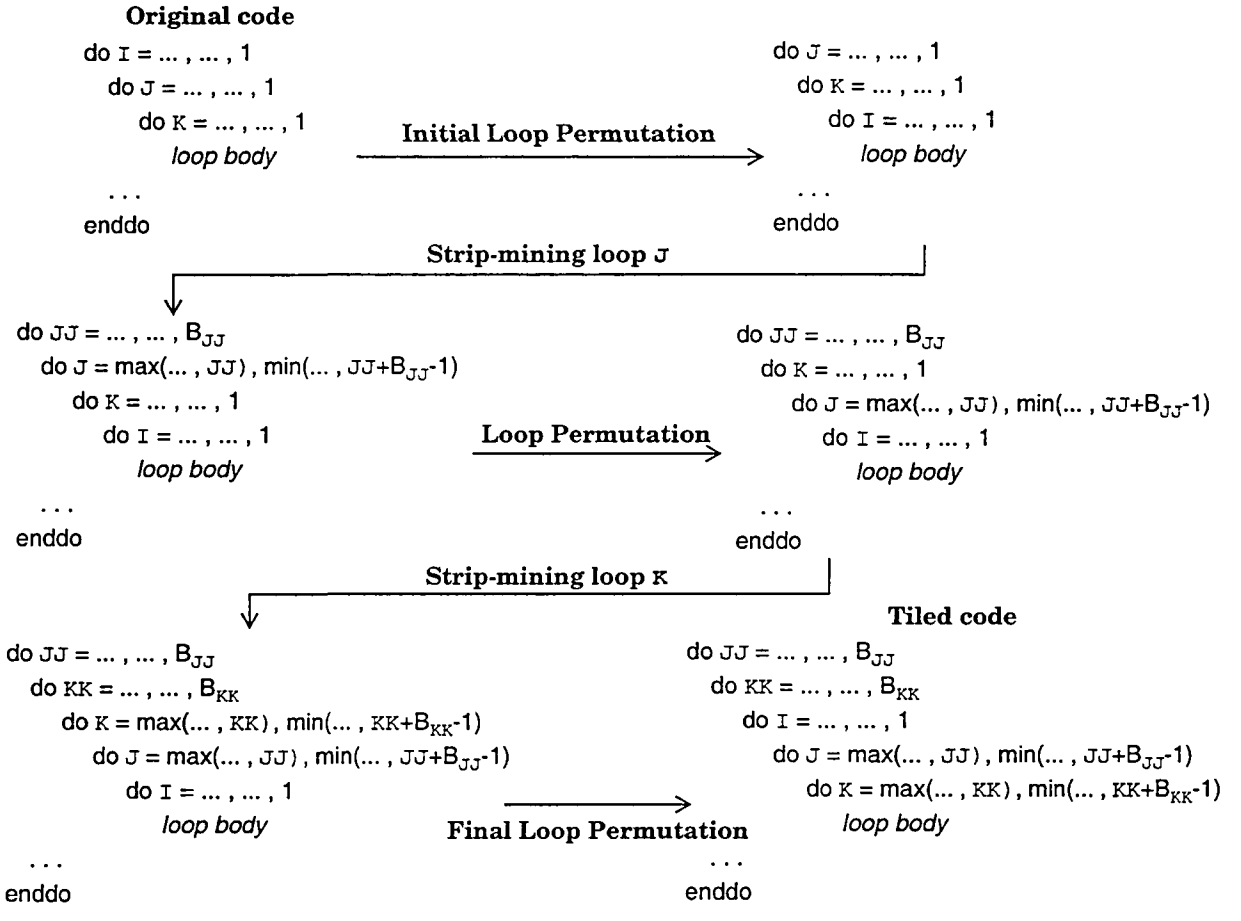

Figure 4.1: Conventional implementation of one level of tiling. We show the order in which strip-mining and loop permutation are applied to partition two dimensions of the 3-dimensional iteration space (two-dimensional tiling). B_{JJ} and B_{II} are the tile sizes in each dimension.

into two loops where the outer loop steps between tiles and the inner loop traverses the points within a tile. After strip-mining, a loop permutation is performed to order the inner loops (the loops that traverse the points within a tile) such that the next loop to be strip-mined becomes the outermost of them. After strip-mining all desired loops, a final loop permutation is required to order the inner loops as desired for the final code. In the final tiled code, the outer loops are the loops that step between tiles (from now on, we will refer to them as *tile-loops* or TI-loops for short) and the inner loops are the loops that traverse the points within the tiles (from now on, we will refer to them as *element loops* or EL-loops[5] for short). For further details refer to Chapter 2, Section 2.3.3.

The loop bounds after strip-mining are directly obtained by applying the formula of strip-mining (see Fig. 4.5 on page 134). The loop bounds after a loop permutation can be obtained using the theory of unimodular transformations [13][121], since the loops involved in the permutation are always loops

5.In Chapter 3, we distinguished between the non-tiled loop and the *element loops* that traverse the iterations inside the register tiles. In this chapter, however, we refer to the non-tiled loop as an EL-loop.

that have steps equal to 1 and, therefore, they define a convex iteration space. To compute the exact bounds, the Fourier-Motzkin Elimination algorithm is used when the loop permutation unimodular matrix is applied [121][129]. Finally, recall that it is not necessary to rewrite the loop body because (1) the strip-mining transformation does not modify the loop body and (2) although the loop permutation does, we use in the transformed code the same names for the loop iteration variables as in the original code.

Multilevel tiling has been implemented by applying tiling level by level [28][121], going from the outermost (i.e., parallelism) to the innermost level (i.e., register level). As an example, in Fig. 4.1 another level of tiling can be performed by applying tiling again to loops `I`, `J` and `K` of the resulting code.

4.2.2 Complexity

The most expensive steps of conventional implementations of tiling are the loop permutation transformations or, more precisely, the steps needed to compute the exact bounds using the Fourier-Motzkin Elimination algorithm (FM algorithm) [7][16][77]. To better clarify the context of this section we first briefly review how the Fourier-Motzkin algorithm is applied to compute the exact bounds and we also review its complexity. The FM algorithm is explained in Appendix A.

Fourier-Motzkin Elimination Algorithm

Let *lp* and *bd* be the number of loops and simple bounds in a loop nest before applying a loop permutation transformation, respectively. The FM algorithm is an algorithm that iterates *lp-1* times[6] and computes the exact loop bounds in the transformed code from innermost loop to outermost loop. In each iteration of FM, two different steps are performed:

- In the first step, all simple bounds of the yet-to-be-processed loops are examined. All simple bounds that are affine functions of the loop iteration variable being solved become simple bounds of this loop.
- In the second step, each of the lower simple bounds of the iteration variable solved in the first step is compared with each of the upper simple bounds. These comparisons generate inequalities that might become new simple bounds of the yet-to-be-processed loops. Note that in the second step of the FM algorithm, the number of simple bounds in each iteration can grow quadratically in the worst case [129].

Let's now examine the complexity of the FM algorithm. In the worst case, the first loop iteration variable to be solved is involved in the *bd* simple bounds of the loop nest. After the first step of FM, the loop iteration variable can have *bd*/2 lower simple bounds and *bd*/2 upper simple bounds. Comparing

6.The bounds of the outermost loop are obtained directly after the FM algorithm has finished.

each of the lower simple bounds with each of the upper simple bounds may give rise to $(bd/2)^2$ new inequalities. If half of the $(bd/2)^2$ new inequalities were lower simple bounds of the next loop iteration variable to be solved and the other half were upper simple bounds, comparing all of them would result in $(bd^2/8)^2$ inequalities. Therefore, since all bd simple bounds could potentially involve all lp iteration variables, the complexity of the FM algorithm is:

$$C_{FM} = O\left(\frac{bd^{2^{(lp-1)}}}{2^{\left(2^{lp}-2\right)}}\right) \approx O\left(\left(\frac{bd}{2}\right)^{2^{(lp-1)}}\right)$$

Thus, the complexity of the FM algorithm depends doubly exponentially on the number of loops involved in the permutation.

Complexity of Conventional Implementations of Multilevel Tiling

We now turn to the cost analysis of conventional implementations of multilevel tiling. Let n and q be the number of loops and simple bounds in the original loop nest, respectively, and let m be the number of loops in the code after multilevel tiling.

A conventional implementation of multilevel tiling executes $(m\text{-}n)$ times the FM algorithm on a set of n loops (the n innermost loops). In the worst case, the number of simple bounds involved the first time the FM algorithm is executed is $q+2$ (q simple bounds of the original code and 2 simple bounds introduced by the previously applied strip-mining). Thus, after the loop permutation, the n innermost EL-loops together can potentially have q_1 simple bounds, where:

$$q_1 = (q+2) + \left(\frac{(q+2)^2}{2^2}\right) + \ldots + \frac{(q+2)^{2^{n-1}}}{2^{2^n-2}} \approx \frac{q^{2^{n-1}}}{2^{2^n-2}} \approx \left(\frac{q}{2}\right)^{2^{n-1}}$$

The next execution of the FM algorithm will deal with $q_1 + 2 \approx q_1$ simple bounds. Note that each time the FM algorithm is executed, the total number of simple bounds in the n innermost loops together could increase, in the worst case, doubly exponentially. Therefore, the complexity of conventional multilevel tiling techniques is:

$$O\left(\left(\frac{q}{2}\right)^{2^{n-1}} + \left(\frac{q_1}{2}\right)^{2^{n-1}} + \ldots + \left(\frac{q_{(m-n-1)}}{2}\right)^{2^{n-1}}\right) \approx O\left(\left(\frac{q}{2}\right)^{2^{n-1}} + \left(\frac{q}{2}\right)^{2^{2n-2}} + \ldots + \left(\frac{q}{2}\right)^{2^{(m-n)n-(m-n))}}\right)$$

Rounding off this expression to the complexity of the last execution of the FM algorithm, it can be expressed by the following formula:

$$C_{conv} = O\left(\left(\frac{q}{2}\right)^{2^{(m-n)(n-1)}}\right)$$

Note that the complexity of conventional multilevel tiling depends doubly exponentially not only on the number of loops involved in the loop permutation, but also on the number of times the FM algorithm is executed (number of TI-loops in the final code).

4.3 SIMULTANEOUS MULTILEVEL TILING

In this section we will show how multilevel tiling can be implemented to deal with all levels simultaneously. The idea behind our Simultaneous Multilevel Tiling algorithm (SMT) consists in applying first strip-mining to all loops at all levels and, afterwards, performing once a single loop permutation transformation to obtain the desired order of the loops.

After applying strip-mining to all loops at all levels, we will obtain a new loop nest that describes a non-convex iteration space. Then, we want to apply a loop permutation transformation to this new loop nest. Although there has been much work [45][85][87][105] addressing the problem of rewriting in a systematic way a loop nest according to a non-singular transformation[7], we cannot make a direct use of these non-singular transformation theories, because they always assume that the source iteration space is a convex space. In our case, however, the source iteration space is non-convex. In this section, we will describe a method to obtain the transformed iteration space when applying a loop permutation transformation to the non-convex space obtained after applying strip-mining to all loops at all levels.

This section is organized as follows: First, we present the framework where we develop our technique. Second, we show how strip-mining is applied to all loops at all levels. Third, we will describe the method needed to obtain the transformed iteration space when applying the loop permutation transformation to the non-convex space. Fourth, we summarize all steps performed by the SMT algorithm and, finally we give an example to illustrate the whole transformation process.

4.3.1 Framework

As we have already explained in Chapter 2, the set of iterations determined by the bounds of n nested loops is a convex subset of Z^n, and we will refer to it as BIS (Bounded Iteration Space):

$$\text{BIS} = \{\vec{I} = (I_1, ..., I_n)^t \mid (L_1 \le I_1 \le U_1, ..., L_n \le I_n \le U_n)\}$$

where $\vec{I}$ is a n-dimensional vector which represents any single iteration of the n-deep loop nest and L_i (U_i) is the lower (upper) bound of loop I_i. The bounds of the loops are max or min functions of affine functions of the surrounding loops iteration variables (i.e. they define a non-rectangular iteration space).

The BIS can be specified in a matrix form [54][77] as follows: $A \cdot \vec{I} \le \beta$, where each row of matrix A and vector β has the coefficients of the loop iteration variables and the independent term of each lower or upper simple bound, respectively. The n elements of vector $\vec{I}$ are the loop iteration variables $(I_1, ..., I_n)$.

7.A transformation represented by a matrix T is non-singular if T has an inverse ($|T| \neq 0$).

A transformation, represented by matrix T, maps each iteration $\vec{I}$ of BIS into one iteration $\vec{J}$ of the Bounded Transformed Iteration Space (BTIS):

$$\text{BTIS} = \{\vec{J} = T \cdot \vec{I} \mid \vec{I} \in \text{BIS}\}$$

The Minimum Convex Space (MCS) which contains all the points of the BTIS can be put in matrix form, using the transformation matrix T and the matrix inequality which represents the bounds of the BIS:

$$\left.\begin{array}{l} A \cdot \vec{I} \leq \beta \\ T \cdot \vec{I} = \vec{J} \end{array}\right\} \quad \Rightarrow \quad \begin{array}{c} A \cdot T^{-1} \cdot \vec{J} \leq \beta \\ \hat{A} \cdot \vec{J} \leq \beta \end{array}$$

The exact bounds of the MCS can be extracted from the matrix inequality $\hat{A} \cdot \vec{J} \leq \beta$, using the Fourier-Motzkin algorithm [16][77].

When T is unimodular, all the integer points of the BTIS have an integer antiimage in the BIS. Therefore, the transformed loop nest must scan all the integer points of the BTIS. In this case, the bounds of the MCS can be directly used to build the loop nest required to scan the BTIS.

However, when T is non-unimodular (its determinant is different from ± 1) there are holes (integer points without integer antiimage) in the BTIS and, in particular, in the boundaries of the BTIS. In this case, to scan correctly the BTIS, all these holes must be skipped. In particular, the bounds of the MCS obtained through the FM algorithm must be corrected to obtain the precise bounds of the BTIS. As an example, consider the loop nest shown in Fig. 4.2a and its corresponding BIS (Fig. 4.2b). After applying the non-unimodular transformation T (Fig. 4.2c) to BIS, we obtain the BTIS shown in Fig. 4.2d.

The grey lines represent the bounds of the MCS and the continuous black lines that bound the shadowed area are the precise bounds of BTIS. Finally, the black dots are the points of BTIS with integer antiimage in BIS.

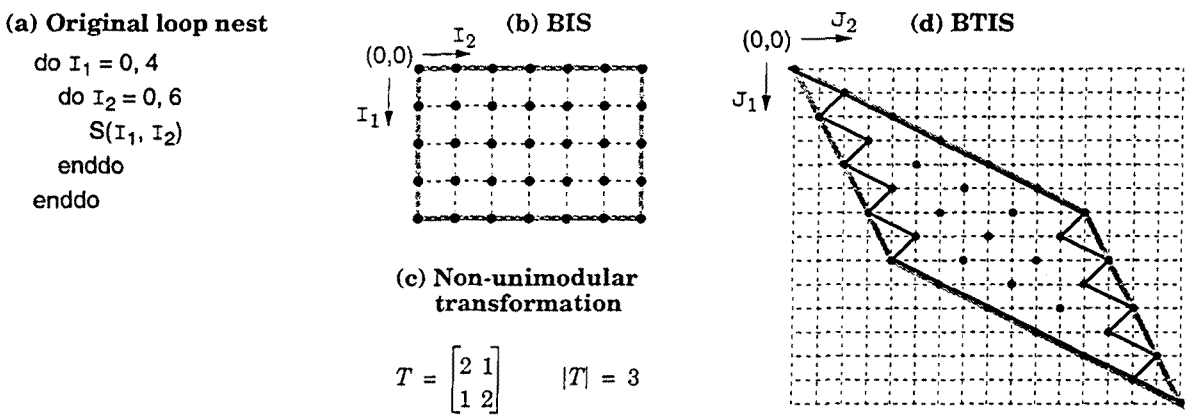

Figure 4.2: (a) Example loop nest. (b) Bounded Iteration Space (BIS) for this loop nest. (c) Non-unimodular transformation T. (d) BTIS obtained after applying the non-unimodular transformation T to BIS. The black dots represent the points in BTIS with integer antiimage in BIS.

Fernández et al. [45] address the problem of correcting in a systematic way the bounds of the MCS given by the FM algorithm, in order to produce the precise bounds of the BTIS. To characterize the BTIS, they use the Hermite Normal Form H of the transformation matrix T [111]. Both H and T generate the same lattice in Z^n. Since H is lower triangular, it permits an easy characterization of the BTIS. The holes of the BTIS are skipped using steps greater than 1 in the loops. These steps are just the elements on the diagonal of matrix H. And the precise bounds of the BTIS are obtained by combining the bounds of the MCS obtained through the FM algorithm with some special non-linear functions [44][115] that involve the non-diagonal elements of H.

Let L_i^T and U_i^T $(1 \le i \le n)$ be the bounds of the MCS of the BTIS, obtained by the FM algorithm, along dimension J_i and let h_{ij} and h'_{ij} be the elements of H and H^{-1} respectively. Then, the transformed code is:

do $J_1 = \lceil L_1^T / h_{11} \rceil \cdot h_{11}, U_1^T, h_{11}$

$gap_2 = (h_{21} \cdot h'_{11} \cdot I_1) \bmod h_{22}$

do $J_2 = \lceil (L_2^T - gap_2) / h_{22} \rceil \cdot h_{22} + gap_2, U_2^T, h_{22}$

...

$$gap_n = \left(\sum_{r=1}^{n-1} h_{nr} \cdot \sum_{d=1}^{r} h'_{rd} \cdot I_d \right) \bmod h_{nn}$$

do $J_n = \lceil (L_n^T - gap_n) / h_{nn} \rceil \cdot h_{nn} + gap_n, U_n^T, h_{nn}$

loop body

enddo

and the transformation of the loop body only requires I_i $(1 \le i \le n)$ (the loop iteration variables in the original loop nest) be replaced by the appropriate linear combination of J_i $(1 \le i \le n)$ (the loop iteration variables for the transformed loop nest); that is:

$$\begin{bmatrix} I_1 \\ \dots \\ I_n \end{bmatrix} = T^{-1} \cdot \begin{bmatrix} J_1 \\ \dots \\ J_n \end{bmatrix}$$

Figure 4.3 shows the transformed code of the example of Fig. 4.2, the Hermite Normal Form H of the transformation matrix T and its inverse H^{-1}.

(a) Hermite Normal form of T

$$H = \begin{bmatrix} 1 & 0 \\ 2 & 3 \end{bmatrix} \qquad H^{-1} = \frac{1}{3} \cdot \begin{bmatrix} 3 & 0 \\ -2 & 1 \end{bmatrix}$$

(b) Transformed loop nest

do $J_1 = 0, 14$

$gap_2 = (2 \cdot J_1) \bmod 3$

do $J_2 = \left\lceil \left(max\left(2 \cdot J_1 - 12, \left\lceil \frac{J_1}{2} \right\rceil\right) - gap_2 \right) / 3 \right\rceil \cdot 3 + gap_2, min\left(2 \cdot J_1, \left\lfloor \frac{18 + J_1}{2} \right\rfloor\right), 3$

S($(2 \cdot J_1 - J_2)/3$, $(2 \cdot J_2 - J_1)/3$)

enddo

Figure 4.3: (a) Hermite Normal Form H (and its inverse H^{-1}) of the transformation matrix T used in Fig. 4.2. (b) Transformed loop nest.

Summarizing, the result of applying a non-unimodular transformation to a convex iteration space is a non-convex space. To obtain the precise bounds of the non-convex transformed space two different steps have to be performed. First, the bounds of the MCS of the transformed space are computed using the Fourier-Motzkin algorithm and, second, these bounds are corrected using the Hermite Normal Form (HNF) of the transformation matrix [111]. Figure 4.4 shows a diagram of the steps performed when a non-unimodular transformation is applied.

In the next subsections, we will show how Simultaneous Multilevel Tiling can be performed using the theory of non-unimodular transformations and we will give an example to illustrate it. As already mentioned, Simultaneous Multilevel Tiling consists of applying first strip-mining to all loops at all levels and, afterwards, performing the loop permutation transformation only once to obtain the desired order of the loops. Let's see how both steps are carried out.

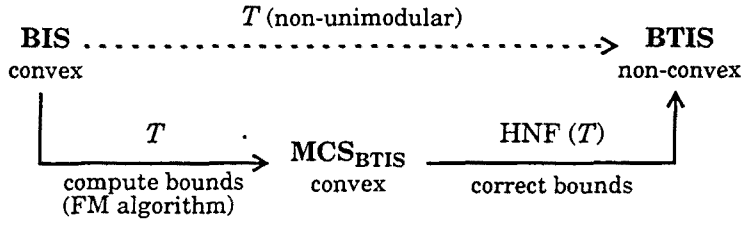

Figure 4.4: Steps performed when a non-unimodular transformation T is applied. HNF(T) is the Hermite Normal Form of T.

4.3.2 Strip-mining all Loops at all Levels

Strip-mining is a loop transformation that divides a set of consecutive points of the iteration space into strips. It decomposes a loop into two loops where the outer loop steps between the strips and the inner loop traverses the points within the strips. The strip-mining transformation [129] is defined by the formula of Figure 4.5, where II is the TI-loop that steps between the strips, I is the EL-loop that traverses the points within the strips, B_{II} is the strip size and $oft_{II} \in Z$ ($0 \leq oft_{II} < B_{II}$) is an offset that determines the origin of the first strip [129]. Using this formula, the tile boundaries are always parallel to the iteration space axes.

$$\text{do I = L, U} \xrightarrow{\textbf{strip-mining}} \begin{array}{l} \text{do II} = \lfloor (L - oft_{II}) / B_{II} \rfloor * B_{II} + oft_{II},\ U,\ B_{II} \\ \text{do I} = \max(II, L),\ \min(II+B_{II}-1,\ U) \end{array}$$

Figure 4.5: Formula for strip-mining a loop.

To strip-mine one loop at several levels, strip-mining is applied repeatedly to the inner EL-loop resulting from the previous round of strip-mining, going from the outermost level to the innermost level. In Fig. 4.5 strip-mining would be applied again to loop I in the resulting code. Note that strip-mining is always applied to loops with a step equal to one. Finally, recall that the strip-mining transformation does not modify the loop body of a loop nest.

As an example, Figure 4.6 shows how strip-mining is applied twice to loop I; it shows the code of the nested loops and the points traversed by loops III, II and I after strip-mining at each level. The points on each line indicate the values of the loop index variables. The strip sizes at each level are B_{III}=11 and B_{II}=4 and, for simplicity, we assume null offsets (oft_{III}=oft_{II}=0). The shadowed rectangles indicate the values of a loop index variable for fixed values of outer loop indices. Note that for a fixed value of loop indices III and II, loop I always iterates inside the tiles determined by them, that is III ≤ I ≤ III+B_{III} -1 and II ≤ I ≤ II+B_{II} -1 always hold. However, for a fixed value of III, loop II can iterate over some points outside the tile determined by III. In particular, III ≤ II does not hold, if B_{III} is not multiple of B_{II}. Notice also that loop index II can iterate over the same point for different values of III.

4.3.3 Loop Permutation

The iteration space defined by the original loop nest is a convex subset of Z^n, that we call the Bounded Iteration Space (BIS). After applying strip-mining to all loops at all levels, we obtain a new loop nest that describes a non-convex iteration space. We will refer to this non-convex iteration space as NCBIS (Non-Convex Bounded Iteration Space). At this point, we want to apply to NCBIS a unimodular loop permutation transformation (T^P) [121] to obtain the Bounded Transformed Iteration Space (BTIS). The BTIS is the desired multilevel tiled iteration space (see Fig. 4.7).

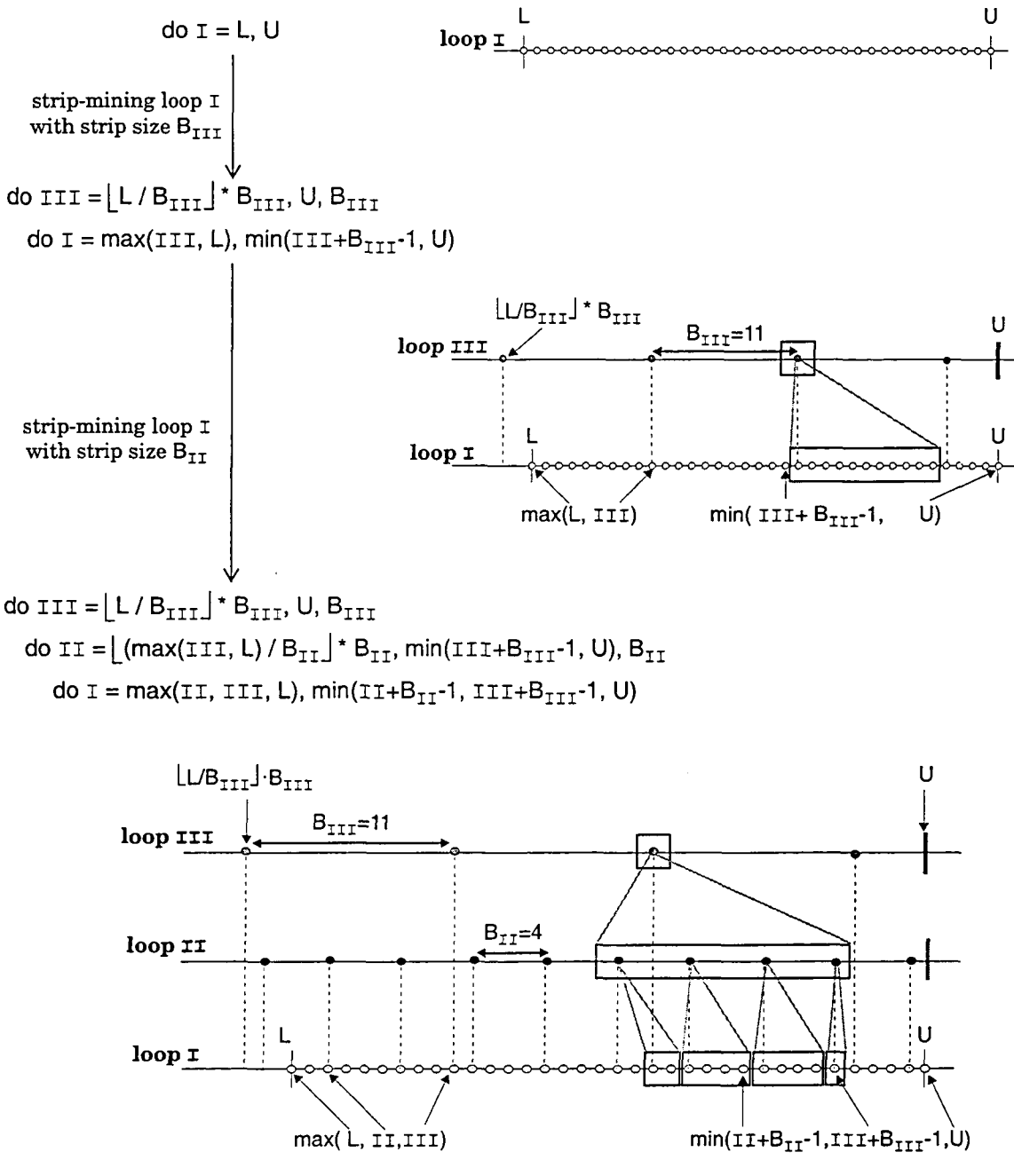

Figure 4.6: How strip-mining is applied to loop I at two levels. The strip sizes at the outermost and innermost level are B_{III}=11 and B_{II}=4, respectively. We assume a null offset.

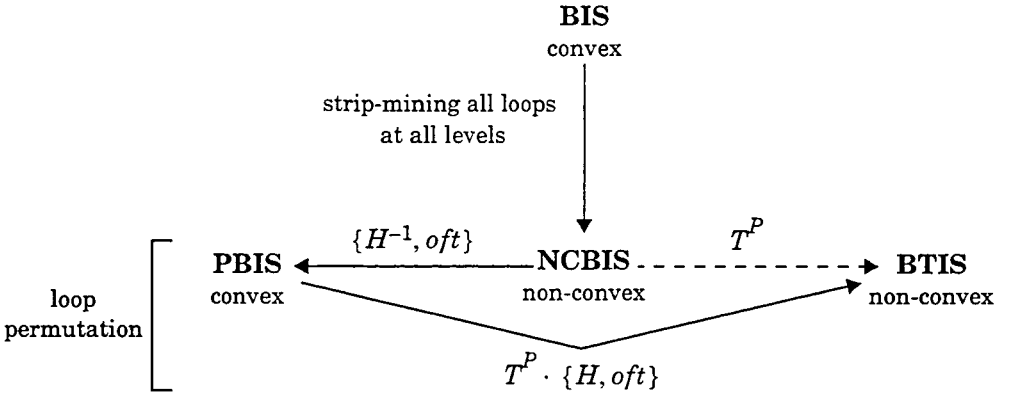

Figure 4.7: Diagram of the Simultaneous Multilevel Tiling transformation process.

There has been much work [45][85][87][105] addressing the problem of rewriting in a systematic way a loop nest according to a non-singular transformation. All these works assume that the source iteration space, to which the transformation is applied, is a convex space. In our case, however, the source iteration space (NCBIS) is non-convex and hence, the theory of non-singular transformations is not directly applicable. The NCBIS is non-convex because some of the loops (the TI-loops) have non-unit steps, and therefore, there are holes in the iteration space. In this section, we will describe a method to obtain the transformed iteration space (BTIS) when applying a unimodular loop permutation transformation to the non-convex space NCBIS.

The development of our work is based on the characterizations of the space NCBIS and of a general space obtained after applying a non-unimodular transformation to a convex space [45]. The expressions of the bounds of the loops that traverse both spaces (NCBIS and a general transformed space) are similar. This fact allows us to deduce a transformation $\{H^{-1}, oft\}$ that transforms NCBIS to a convex space. We will refer to this convex iteration space as PBIS (Previous Bounded Iteration Space). In fact, H^{-1} is a semi-normalization transformation [129] that makes all loops in PBIS have a step equal to one and oft is an offset. Next, the multilevel tiled iteration space BTIS can be obtained by applying a non-unimodular transformation $T = T^P \cdot \{H, oft\}$ to PBIS. This transformation performs the loop permutation and undoes the previous semi-normalization applied to NCBIS. Figure 4.7 shows a diagram of the Simultaneous Multilevel Tiling (SMT) transformation process.

In the remainder of this section we will show how the loop bounds of the loop nest that traverses BTIS can be computed in a systematic way and, at the end of this section, we will show that the loop body does not need to be rewritten. For simplicity, from now on, we will use the abbreviations NCBIS, PBIS and BTIS to refer, indistinctly, to the iteration space or to the loop nest that traverses the iteration space.

Loop Bounds Computation

We start by giving some definitions to characterize the spaces NCBIS and PBIS and we present a lemma that shows how NCBIS can be obtained from the convex space PBIS. Then, we give two corollaries showing how the bounds of the Minimum Convex Space of NCBIS can be computed. Finally, we enounce a theorem that allows us to compute the bounds of BTIS in a systematic way.

In this section, we use a slightly different notation with respect to previous sections. Unless it were necessary for readability, from now on we will not use any more the arrows above the vectors representing loop iteration variables as we have been doing so far.

Definition 1

Let H and *oft* be an mxm diagonal matrix and a m-dimensional column vector, respectively:

$$H = \begin{bmatrix} \mathsf{B}_1 & & \varnothing \\ & \cdots & \\ \varnothing & & \mathsf{B}_m \end{bmatrix} \qquad oft = \begin{bmatrix} \mathsf{oft}_1 \\ \cdots \\ \mathsf{oft}_m \end{bmatrix}$$

where $\mathsf{oft}_k, \mathsf{B}_k \in \mathsf{Z}$ and $0 \le \mathsf{oft}_k < \mathsf{B}_k$ $(1 \le k \le m)$.

Definition 2

The m-deep loop nest obtained after the strip-mining phase defines the m-dimensional non-convex space NCBIS and a loop $\mathtt{I}_k^{NCBIS}$ in NCBIS can be written, in the general case, into the following form:

$$\text{do } \mathtt{I}_k^{NCBIS} = \left\lfloor \left(\mathsf{L}_k^{NCBIS} - \mathsf{oft}_k \right) / \mathsf{B}_k \right\rfloor \cdot \mathsf{B}_k + \mathsf{oft}_k, \mathsf{U}_k^{NCBIS}, \mathsf{B}_k \qquad (1)$$

where

- B_k is the kth diagonal element of a matrix H,
- oft_k is the kth element of a vector *oft,* and
- L_k^{NCBIS} and U_k^{NCBIS} have the forms $max(l_{k,0}, l_{k,1}, l_{k,2}, \ldots)$ and $min(u_{k,0}, u_{k,1}, u_{k,2}, \ldots)$, respectively, and $l_{k,j}$, $u_{k,j}$ are linear functions of the loops iteration variables, that can be expressed as follows:

$$l_{k,j} = f_{k,j}^{l}\left(\mathtt{I}_1^{NCBIS}, \ldots, \mathtt{I}_m^{NCBIS} \right) = f_{k,j}^{l}(\mathtt{I}^{NCBIS})$$
$$u_{k,j} = f_{k,j}^{u}\left(\mathtt{I}_1^{NCBIS}, \ldots, \mathtt{I}_m^{NCBIS} \right) = f_{k,j}^{u}(\mathtt{I}^{NCBIS})$$

, where $\mathtt{I}^{NCBIS} = \begin{bmatrix} \mathtt{I}_1^{NCBIS} \\ \cdots \\ \mathtt{I}_m^{NCBIS} \end{bmatrix}$

By construction, a loop $\mathtt{I}_k$ with step equal to one ($\mathsf{B}_k = 1$) always has a null offset ($\mathsf{oft}_k = 0$).

Definition 3

A loop I_k^{PBIS} of the m-deep loop nest that defines the convex iteration space PBIS can be written, in the general case, into the following form:

$$\text{do } I_k^{PBIS} = \left\lfloor \left(L_k^{PBIS} - \text{oft}_k \right) / B_k \right\rfloor, \left\lfloor \left(U_k^{PBIS} - \text{oft}_k \right) / B_k \right\rfloor \qquad (2)$$

where

- B_k is the kth diagonal element of a matrix H,
- oft_k is the kth element of a vector *oft*, and
- L_k^{PBIS} and U_k^{PBIS} have the forms $max(\tilde{l}_{k,0}, \tilde{l}_{k,1}, \tilde{l}_{k,2}, \ldots)$ and $min(\tilde{u}_{k,0}, \tilde{u}_{k,1}, \tilde{u}_{k,2}, \ldots)$, respectively, and

$$\tilde{l}_{k,j} = f_{k,j}^{l}(H \cdot I^{PBIS} + oft)$$
$$\tilde{u}_{k,j} = f_{k,j}^{u}(H \cdot I^{PBIS} + oft)$$

, where $I^{PBIS} = \begin{bmatrix} I_1^{PBIS} \\ \ldots \\ I_m^{PBIS} \end{bmatrix}$

and $f_{k,j}^{l}$ and $f_{k,j}^{u}$ are the same linear functions as in NCBIS.

From these definitions one important lemma follows.

Lemma 1

The non-convex iteration space NCBIS (1) can be obtained from the convex iteration space PBIS (2) using the following transformation:

$$I^{NCBIS} = H \cdot I^{PBIS} + oft$$

Proof

Let $A^{PBIS} \cdot I^{PBIS} \leq \beta^{PBIS}$ be the bounds of PBIS specified in matrix form. By applying the transformation ($I^{PBIS} = H^{-1} \cdot \left(I^{NCBIS} - oft \right)$), we obtain the following system of linear inequalities:

$$A^{PBIS} \cdot H^{-1} \cdot I^{NCBIS} - A^{PBIS} \cdot H^{-1} \cdot oft \leq \beta^{PBIS}$$

Let's now add two new inequalities to the system that can be assimilated to a new (outermost) loop that only performs one iteration and its value is always one:

$$I_0 \leq 1 = \beta_0$$
$$-I_0 \leq -1$$

Since $\mathtt{I}_0$ is equal to 1, the new extended system can be written into the following form:

$$\left[\begin{array}{c:c} 1 & 0 \\ \hdashline -1 & \\ \hdashline -A^{PBIS}\cdot H^{-1}\cdot oft & A^{PBIS}\cdot H^{-1} \end{array}\right]\cdot\left[\begin{array}{c} \mathtt{I}_0 \\ \mathtt{I}^{NCBIS} \end{array}\right]\leq\left[\begin{array}{c} \beta_0 \\ -\beta_0 \\ \beta^{PBIS} \end{array}\right]$$

and this system can also be rewritten as follows:

$$\underbrace{\left[\begin{array}{c:c} 1 & 0 \\ \hdashline -1 & \\ \hdashline 0 & A^{PBIS} \end{array}\right]}_{A_E^{PBIS}}\cdot\underbrace{\left[\begin{array}{c:c} 1 & 0 \\ \hdashline -H^{-1}\cdot oft & H^{-1} \end{array}\right]}_{H_E^{-1}}\cdot\underbrace{\left[\begin{array}{c} \mathtt{I}_0 \\ \mathtt{I}^{NCBIS} \end{array}\right]}_{\mathtt{I}_E^{NCBIS}}\leq\underbrace{\left[\begin{array}{c} \beta_0 \\ -\beta_0 \\ \beta^{PBIS} \end{array}\right]}_{\beta_E^{PBIS}} \qquad (3)$$

Let's refer as $A_E^{PBIS}\cdot H_E^{-1}\cdot \mathtt{I}_E^{NCBIS}\leq\beta_E^{PBIS}$ to the extended system. The matrix transformation H_E is a non-unimodular lower triangular matrix, therefore HNF(H_E)=H_E. Then, using the method proposed in [45] (see Section 4.3.1) to compute the exact bounds of NCBIS, we have that:

(a) The bounds of the MCS of NCBIS are computed by solving the system (3). The two first inequalities give us:

$$\mathtt{I}_0\leq\beta_0$$
$$-\mathtt{I}_0\leq-\beta_0$$

and the remainder inequalities can be written as follows:

$$A^{PBIS}\cdot\left[-H^{-1}\cdot oft \quad H^{-1}\right]\cdot\left[\begin{array}{c} \mathtt{I}_0 \\ \mathtt{I}^{NCBIS} \end{array}\right]\leq\beta^{PBIS}\ \Rightarrow$$

$$A^{PBIS}\cdot\left[-H^{-1}\cdot oft\cdot\mathtt{I}_0+H^{-1}\cdot\mathtt{I}^{NCBIS}\right]\leq\beta^{PBIS} \qquad (4)$$

From Definition 3 we have that the jth (lower or upper) bound of loop $\mathtt{I}_k^{PBIS}$ represented in the system $A^{PBIS}\cdot\mathtt{I}^{PBIS}\leq\beta^{PBIS}$ has the following form:

- if it is a lower bound[8]: $f_{k,j}^{l}(H\cdot\mathtt{I}^{PBIS}+oft)-\mathtt{B}_k\cdot\mathtt{I}_k^{PBIS}-\mathtt{oft}_k-\mathtt{B}_k+1\leq 0$
- if it is an upper bound: $-f_{k,j}^{u}(H\cdot\mathtt{I}^{PBIS}+oft)+\mathtt{B}_k\cdot\mathtt{I}_k^{PBIS}+\mathtt{oft}_k\leq 0$

8.The floor function in the lower bound must be converted in a ceiling function in the following way: $\lfloor a/b\rfloor=\lceil(a-b+1)/b\rceil$

Moreover, the proposed transformation $\mathtt{I}^{PBIS} = H^{-1} \cdot \left(\mathtt{I}^{NCBIS} - oft \right)$ of Lemma 1 can be rewritten in the extended space as follows, because $\mathtt{I}_0 = \beta_0 = 1$:

$$\mathtt{I}^{PBIS} = H^{-1} \cdot \left(\mathtt{I}^{NCBIS} - oft \cdot \mathtt{I}_0 \right), \text{ that is, } \mathtt{I}_k^{PBIS} = \frac{1}{\mathtt{B}_k} \cdot \left(\mathtt{I}_k^{NCBIS} - \mathtt{oft}_k \cdot \mathtt{I}_0 \right) \qquad (1 \le k \le m)$$

Then, it can be deduced after some algebraic manipulations, that the jth (lower or upper) bound along dimension $\mathtt{I}_k^{NCBIS}$ ($(1 \le k \le m)$) of the system (4) has the following form:

- if it is a lower bound: $\mathtt{I}_k^{NCBIS} \ge f_{k,j}^{l}\left(\mathtt{I}^{NCBIS} \right) - \mathtt{B}_k + 1 = l_{k,j} - \mathtt{B}_k + 1$
- if it is an upper bound: $\mathtt{I}_k^{NCBIS} \le f_{k,j}^{u}\left(\mathtt{I}^{NCBIS} \right) = u_{k,j}$

Therefore, the solution to the system (3) is:

$$\mathtt{L}_0^{MCS} = \mathtt{U}_0^{MCS} = 1$$
$$\mathtt{L}_k^{MCS} = \mathtt{L}_k^{NCBIS} - \mathtt{B}_k + 1$$
$$\mathtt{U}_k^{MCS} = \mathtt{U}_k^{NCBIS}$$

where $\mathtt{L}_k^{MCS}$ and $\mathtt{U}_k^{MCS}$ $(0 \le k \le m)$ are the lower and upper bounds of the MCS of NCBIS along dimension $\mathtt{I}_k^{NCBIS}$, respectively.

(b) The HNF of the extended matrix H_E is used to correct the bounds of the MCS of NCBIS, where:

$$HNF(H_E) = H_E = \begin{bmatrix} 1 & 0 \\ -H^{-1} \cdot oft & H^{-1} \end{bmatrix}^{-1} = \begin{bmatrix} 1 & 0 \\ oft & H \end{bmatrix}$$

Then, the exact loop bounds in NCBIS are:

do $\mathtt{I}_0 = 1, 1$

...

do $\mathtt{I}_k^{NCBIS} = \left\lceil \dfrac{\mathtt{L}_k^{NCBIS} - \mathtt{B}_k + 1 - ((\mathtt{oft}_k \cdot \mathtt{I}_0) \bmod \mathtt{B}_k)}{\mathtt{B}_k} \right\rceil \cdot \mathtt{B}_k + ((\mathtt{oft}_k \cdot \mathtt{I}_0) \bmod \mathtt{B}_k), \mathtt{U}_k^{NCBIS}, \mathtt{B}_k$

Now, loop $\mathtt{I}_0$ can be eliminated and the loop nest can be rewritten into the following form:

$$\text{do } \mathtt{I}_k^{NCBIS} = \left\lfloor \frac{\mathtt{L}_k^{NCBIS} - \mathtt{oft}_k}{\mathtt{B}_k} \right\rfloor \cdot \mathtt{B}_k + \mathtt{oft}_k, \mathtt{U}_k^{NCBIS}, \mathtt{B}_k \qquad (1 \le k \le m)$$

bearing in mind that $\mathtt{I}_0 = 1$, $\mathtt{oft}_k < \mathtt{B}_k$ (therefore, $(\mathtt{oft}_k \cdot \mathtt{I}_0) \bmod \mathtt{B}_k = \mathtt{oft}_k$) and $\left\lceil \dfrac{a-b+1}{b} \right\rceil = \left\lfloor \dfrac{a}{b} \right\rfloor$.

This is the general form of a loop in NCBIS (Definition 2). ■

Let's now see the two corollaries that show how the bounds of the MCS of NCBIS can be computed.

Corollary 1

The bounds of the Minimum Convex Space (MCS) of the m-dimensional non-convex space NCBIS can be obtained from the following system of inequalities:

$$A^{NCBIS} \cdot \mathtt{I}^{NCBIS} \leq \beta^{NCBIS}$$

where $A^{NCBIS} = A^{PBIS} \cdot H^{-1}$ and $\beta^{NCBIS} = \beta^{PBIS} + A^{PBIS} \cdot H^{-1} \cdot oft$.

Proof

From Lemma 1 we know that the following system represents the MCS of NCBIS:

$$A^{PBIS} \cdot \left[-H^{-1} \cdot oft \cdot \mathtt{I}_0 + H^{-1} \cdot \mathtt{I}^{NCBIS}\right] \leq \beta^{PBIS}$$

Substituting $\mathtt{I}_0 = 1$ and rewriting the expression we obtain:

$$A^{PBIS} \cdot H^{-1} \cdot \mathtt{I}^{NCBIS} \leq \beta^{PBIS} + A^{PBIS} \cdot H^{-1} \cdot oft \quad \Rightarrow \quad A^{NCBIS} \cdot \mathtt{I}^{NCBIS} \leq \beta^{NCBIS} \ \blacksquare$$

Corollary 2

The lower and upper bounds of the MCS of NCBIS along dimension $\mathtt{I}_k^{NCBIS}$ ($\mathtt{L}_k^{MCS}$ and $\mathtt{U}_k^{MCS}$, respectively) can be directly obtained from the loop bounds of NCBIS (Definition 2) as follows:

$$\begin{aligned} \mathtt{L}_k^{MCS} &= \mathtt{L}_k^{NCBIS} - \mathtt{B}_k + 1 \\ \mathtt{U}_k^{MCS} &= \mathtt{U}_k^{NCBIS} \end{aligned} \qquad (1 \leq k \leq m)$$

Proof

See proof of Lemma 1. ■

Until now, we have proven that NCBIS can be obtained from the convex iteration space PBIS by applying a particular transformation and we have shown how the bounds of the MCS of NCBIS can be directly obtained from the loop bounds of NCBIS. Now we enounce a theorem that allows us to compute the bounds of BTIS in a systematic way.

Theorem 1

Let BTIS be the transformed iteration space obtained after applying a unimodular loop permutation transformation T^P to the m-dimensional space NCBIS. The m-deep loop nest that traverses BTIS has the following form:

$$\text{do } \mathtt{I}_k^{BTIS} = \left\lceil \left(\mathsf{L}_k^{BTIS} - \mathsf{oft}_k^P \right) / \mathsf{B}_k^P \right\rceil \cdot \mathsf{B}_k^P + \mathsf{oft}_k^P, \mathsf{U}_k^{BTIS}, \mathsf{B}_k^P \qquad (5)$$

where

- L_k^{BTIS} and U_k^{BTIS} $(1 \le k \le m)$ are obtained by solving the following system using the Fourier-Motzkin algorithm:

$$A^{NCBIS} \cdot \left(T^P \right)^{-1} \cdot \mathtt{I}^{BTIS} \le \beta^{NCBIS} \text{ , where } \mathtt{I}^{BTIS} = \begin{bmatrix} \mathtt{I}_1^{BTIS} \\ \cdots \\ \mathtt{I}_m^{BTIS} \end{bmatrix}$$

- oft_k^P is the kth elements of a vector $oft^P = T^P \cdot oft$,
- and B_k^P is the kth diagonal element of a matrix $H^P = T^P \cdot H \cdot \left(T^P \right)^{-1}$.

Proof

From Lemma 1 we have that NCBIS can be obtained from the convex iteration space PBIS $A^{PBIS} \cdot \mathtt{I}^{PBIS} \le \beta^{PBIS}$ using the transformation $\mathtt{I}^{NCBIS} = H \cdot \mathtt{I}^{PBIS} + oft$.

Let's now extend PBIS in one dimension ($\mathtt{I}_0$) that represents a new (outermost) loop that only performs one iteration and its value is always one ($\beta_0 = 1$). The new extended system can be written in matrix form into the following manner:

$$\underbrace{\begin{bmatrix} 1 & 0 \\ -1 & \\ 0 & A^{PBIS} \end{bmatrix}}_{A_E^{PBIS}} \cdot \underbrace{\begin{bmatrix} \mathtt{I}_0 \\ \mathtt{I}^{PBIS} \end{bmatrix}}_{\mathtt{I}_E^{PBIS}} \le \underbrace{\begin{bmatrix} \beta_0 \\ -\beta_0 \\ \beta^{PBIS} \end{bmatrix}}_{\beta_E^{PBIS}}$$

As shown in proof of Lemma 1, the transformation matrix that transforms the extended PBIS into the extended NCBIS is:

$$H_E = \begin{bmatrix} 1 & 0 \\ oft & H \end{bmatrix}$$

Then, the matrix transformation that transforms the extended PBIS into an extended BTIS is $\mathtt{I}_E^{BTIS} = T_E^P \cdot H_E \cdot \mathtt{I}_E^{PBIS}$, where:

$$T_E^P \cdot H_E = \begin{bmatrix} 1 & 0 \\ 0 & T^P \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ oft & H \end{bmatrix}, \text{ and } \mathtt{I}_E^{BTIS} = \begin{bmatrix} \mathtt{I}_0 \\ \mathtt{I}^{BTIS} \end{bmatrix}$$

By applying this transformation to the extended PBIS the following system is obtained:

$$A_E^{PBIS} \cdot H_E^{-1} \cdot \left(T_E^P\right)^{-1} \cdot \mathtt{I}_E^{BTIS} \leq \beta_E^{PBIS} \qquad (6)$$

To compute the exact bounds of the extended BTIS, we use the method proposed in [45]: first the bounds of the MCS are computed and, second, the bounds are corrected using the Hermite Normal Form of $T_E^P \cdot H_E$.

(a) The bounds of the MCS of the extended BTIS are computed by solving system (6). The two first inequalities give us:

$$\mathtt{I}_0 \leq \beta_0$$
$$-\mathtt{I}_0 \leq -\beta_0$$

and the remainder inequalities can be written as follows:

$$A^{PBIS} \cdot \begin{bmatrix} -H^{-1} \cdot oft & (T^P \cdot H)^{-1} \end{bmatrix} \cdot \begin{bmatrix} \mathtt{I}_0 \\ \mathtt{I}^{BTIS} \end{bmatrix} \leq \beta^{PBIS} \Rightarrow$$

$$A^{PBIS} \cdot \left[-H^{-1} \cdot oft \cdot \mathtt{I}_0 + (T^P \cdot H)^{-1} \cdot \mathtt{I}^{NCBIS} \right] \leq \beta^{PBIS} \qquad (7)$$

To rewrite this expression in terms of NCBIS, we use Corollary 1. From Corollary 1 we have that $A^{PBIS} = A^{NCBIS} \cdot H$ and $\beta^{PBIS} = \beta^{NCBIS} - A^{PBIS} \cdot H^{-1} \cdot oft$, therefore, substituting in (7) we have that the bounds of the MCS of BTIS are obtained from the following system using the Fourier-Motzkin algorithm:

$$A^{NCBIS} \cdot H \cdot (T^P \cdot H)^{-1} \leq \beta^{NCBIS} \Rightarrow A^{NCBIS} \cdot (T^P)^{-1} \leq \beta^{NCBIS}$$

(b) The HNF of $T_E^P \cdot H_E$ is used to correct the bounds of the MCS of the BTIS. Since H_E is a lower triangular matrix, $HNF\left(T_E^P \cdot H_E\right) = T_E^P \cdot H_E \cdot \left(T_E^P\right)^{-1}$, that is:

$$HNF\left(T_E^P \cdot H_E\right) = \begin{bmatrix} 1 & 0 \\ 0 & T^P \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ oft & H \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ 0 & \left(T^P\right)^{-1} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ T^P \cdot oft & T^P \cdot H \cdot (T^P)^{-1} \end{bmatrix}, \text{ then}$$

$$HNF\left(T_E^P \cdot H_E\right) = \begin{bmatrix} 1 & 0 \\ oft^P & H^P \end{bmatrix}, \text{ where } oft^P = T^P \cdot oft \text{ and } H^P = T^P \cdot H \cdot \left(T^P\right)^{-1}.$$

Then, the exact loop bounds in BTIS are:

$$\text{do } I_0 = 1, 1$$

$$\cdots$$

$$\text{do } I_k^{BTIS} = \left\lceil \frac{L_k^{BTIS} - \left(\left(oft_k^P \cdot I_0 \right) \bmod B_k^P \right)}{B_k^P} \right\rceil \cdot B_k^P + \left(\left(oft_k^P \cdot I_0 \right) \bmod B_k^P \right), U_k^{BTIS}, B_k^P$$

where L_k^{BTIS} and U_k^{BTIS} $(1 \le k \le m)$ are the bounds of the MCS of BTIS along dimension I_k^{BTIS}, oft_k^P is the kth element of vector oft^P and B_k^P is the kth diagonal element of H^P.

Now, loop I_0 can be eliminated and the loop nest can be rewritten into the following form,

$$\text{do } I_k^{BTIS} = \left\lceil \frac{L_k^{BTIS} - oft_k^P}{B_k^P} \right\rceil \cdot B_k^P + oft_k^P, U_k^{BTIS}, B_k^P \qquad (1 \le k \le m)$$

bearing in mind that $I_0 = 1$ and $oft_k^P < B_k^P$ (therefore, $\left(oft_k^P \cdot I_0 \right) \bmod B_k^P = oft_k^P$). ■

Rewriting the Loop Body

The loop body of the original iteration space (BIS) does not need to be rewritten after the Simultaneous Multilevel Tiling transformation because (1) the strip-mining phase does not modify the loop body and (2) although the loop permutation phase does, we use in the transformed code (BTIS) the same names for the loop iteration variables as in the strip-mined code (NCBIS), thus avoiding a rewrite of the loop body.

Let I^{NCBIS} and I^{BTIS} be the loop iteration variables of NCBIS and BTIS, respectively. The transformation of the loop body after applying the loop permutation transformation T^P to NCBIS, only requires I_k^{NCBIS} $(1 \le k \le m)$ be replaced by the appropriate linear combination of I_k^{BTIS} $(1 \le k \le m)$; that is:

$$\begin{bmatrix} I_1^{NCBIS} \\ \cdots \\ I_m^{NCBIS} \end{bmatrix} = \left(T^P \right)^{-1} \cdot \begin{bmatrix} I_1^{BTIS} \\ \cdots \\ I_m^{BTIS} \end{bmatrix}$$

Since T^P is the identity matrix with rows permuted, each row of T^P has one unit element and zeros elsewhere. Thus, we can name the loop iteration variables in BTIS as in NCBIS according to the permutation, that is $I^{BTIS} = T^P \cdot I^{NCBIS}$. This way, the loop body does not need to be modified.

4.3.4 SMT Summary

Summarizing the previous sections, in order to transform a n-deep loop nest (BIS) into a m-deep multilevel tiled code (BTIS), the following steps have to be performed:

Step 1: Apply strip-mining to all desired loops in BIS at all desired levels, obtaining a m-deep loop nest (NCBIS).

Step 2: Compute matrix H, vector oft and the bounds of the MCS of NCBIS directly from the bounds of NCBIS. Recall from Corollary 2 that L_k^{MCS} and U_k^{MCS} $(1 \le k \le m)$ (the lower and upper bounds of the MCS along dimension $\mathtt{I}_k^{NCBIS}$, respectively) can be directly obtained from the bounds of NCBIS as follows:

$$\mathsf{L}_k^{MCS} = \mathsf{L}_k^{NCBIS} - \mathsf{B}_k + 1$$
$$\mathsf{U}_k^{MCS} = \mathsf{U}_k^{NCBIS}$$

The MCS of NCBIS can be written in matrix form as $A^{NCBIS} \cdot \mathtt{I}^{NCBIS} \le \beta^{NCBIS}$.

Step 3: Compute the bounds of the MCS of the BTIS. They are extracted from the matrix inequality $A^{NCBIS} \cdot \left(T^P\right)^{-1} \cdot \mathtt{I}^{BTIS} \le \beta^{NCBIS}$, using the Fourier-Motzkin algorithm (Theorem 1). T^P is the unimodular loop permutation transformation.

Step 4: Correct the bounds of the MCS of the BTIS using the vector oft^P ($oft^P = T^P \cdot oft$) and the matrix H^P ($H^P = T^P \cdot H \cdot \left(T^P\right)^{-1}$). At this point we have obtained the exact bounds of the BTIS.

4.3.5 SMT Example

To exemplify the transformation process we will use the code shown in Fig. 4.8a and we assume that tiling has to be performed at two levels. Suppose that, at the innermost level, dimensions `I` and `J` have to be tiled and, at the outermost level, only dimension `I` has to be tiled. After strip-mining all loops at all levels, using all offsets equal to 1, we obtain the code shown in Fig. 4.8b (the NCBIS). The length of the loop index identifiers (`III-II-I`) refers to the level of the tiles that they are traversing. Suppose that the desired loop order in the transformed code is `III-JJ-II-J-I`, from outermost to innermost.

Figures 4.8c to 4.8h show the loop permutation process. First, we compute matrix H, vector oft (Fig. 4.8c) and the bounds of the MCS of NCBIS (Fig. 4.8d) directly from the bounds of the NCBIS (Fig. 4.8b). Second, we compute the bounds of the MCS of BTIS (Fig. 4.8f), using the FM algorithm. These bounds are extracted from the matrix inequality $A^{NCBIS} \cdot \left(T^P\right)^{-1} \cdot \mathtt{I}^{BTIS} \le \beta^{NCBIS}$, where $A^{NCBIS} \cdot \mathtt{I}^{NCBIS} \le \beta^{NCBIS}$ are the bounds of the MCS of NCBIS (Fig. 4.8d) and T^P is the unimodular loop permutation transformation (Fig. 4.8e). Finally, we correct the bounds of the MCS of BTIS (Fig. 4.8h) as explained in Section 4.3.1 (page 132), using H^P and oft^P (Fig. 4.8g).

Step 1: Strip-mining phase

(a) BIS

```
do I = 1, N
  do J = I, N
    loop body
enddo
```

strip-mining phase →

(b) NCBIS

```
do III =1, N, B_III
  do II = ⌊max(III-1, 0)/ B_II⌋ * B_II+1, min(III+B_III-1, N), B_II
    do I = max(II, III, 1), min(II+B_II-1, III+B_III-1, N)
      do JJ = ⌊I-1/ B_JJ⌋ * B_JJ+1, N, B_JJ
        do J = max(I, JJ) , min(N, JJ+B_JJ-1)
          loop body
enddo
```

Step 2: Compute *H*, *oft* and MCS of NCBIS

(c) Matrix *H* and vector *oft*

$$H = \begin{bmatrix} B_{III} & & & & \\ & B_{II} & & 0 & \\ & & 1 & & \\ 0 & & & B_{JJ} & \\ & & & & 1 \end{bmatrix} \qquad oft = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

(d) MCS of NCBIS

$$2\text{-}B_{III} \le III \le N$$
$$\max(III\text{-}B_{II}+1,\ 2\text{-}B_{II}) \le II \le \min(III+B_{III}\text{-}1,\ N)$$
$$\max(II,\ III,\ 1) \le I \le \min(II+B_{II}\text{-}1,\ III+B_{III}\text{-}1,\ N)$$
$$I\text{-}B_{JJ}+1 \le JJ \le N$$
$$\max(I,\ JJ) \le J \le \min(N,\ JJ+B_{JJ}\text{-}1)$$

Step 3: Compute MCS of BTIS using T^P

(e) Matrix T^P

$$T^P = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

(f) MCS of BTIS

$$(2\text{-}B_{III}) \le III \le N$$
$$\max(III,\ 1)\text{-}B_{JJ}+1 \le JJ \le N$$
$$\max(III,\ 1)\text{-}B_{II}+1 \le II \le \min(III+B_{III}\text{-}1,\ JJ+B_{JJ}\text{-}1, N)$$
$$\max(III,\ II,\ JJ, 1) \le J \le \min(JJ+B_{JJ}\text{-}1, N)$$
$$\max(III,\ II, 1) \le I \le \min(III+B_{III}\text{-}1,\ II+B_{II}\text{-}1,\ JJ+B_{JJ}\text{-}1,\ J,\ N)$$

Step 4: Correct the bounds using oft^P and matrix H^P

(g) Matrix H^P and vector oft^P

$$H^P = \begin{bmatrix} B_{III} & & & & \\ & B_{JJ} & & 0 & \\ & & B_{II} & & \\ 0 & & & 1 & \\ & & & & 1 \end{bmatrix} \qquad oft^P = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

(h) Exact bounds of BTIS

```
do III =1, N, B_III
  do JJ = ⌊(max(III-1, 0)/ B_JJ⌋*B_JJ+1, N, B_JJ
    do II = ⌊max(III-1,0)/ B_II⌋*B_II+1,
            min(III+B_III-1,JJ+B_JJ-1,N), B_II
      do J = max(III, II, JJ, 1), min(JJ+B_JJ-1, N)
        do I = max(III, II, 1),
               min(III+B_III-1, II+B_II-1, JJ+B_JJ-1, J, N)
          loop body
enddo
```

Figure 4.8: (a) Example of loop nest (BIS). (b) Loop nest after applying strip-mining to loop I at two levels and to J at one level (NCBIS). (c) Matrix *H* and vector *oft*, obtained from NCBIS. (d) Minimum Convex Space of NCBIS, directly obtained from (b). (e) Loop permutation matrix T^P used in the example. (f) Minimum Convex Space of BTIS, obtained using the FM algorithm. (g) Matrix H^P and vector oft^P used to correct the bounds of the MCS of BTIS. (h) Exact loop bounds in the BTIS.

4.4 EFFICIENT IMPLEMENTATION OF SMT

In the previous section we have shown how multilevel tiling can be implemented to deal with all levels simultaneously. Of all the steps required to implement our multilevel tiling technique, the most expensive one is the loop permutation transformation (step 3) or, more precisely, the step that computes the bounds of the MCS of the BTIS using the Fourier-Motzkin algorithm. Recall from Section 4.2.2 that the complexity of the FM algorithm depends doubly exponentially on the number of loops involved in the permutation.

In this section we propose an efficient implementation of SMT that allows us to demonstrate (later in Section 4.5) that the whole SMT process has a much lower complexity than traditional techniques. The idea behind our SMT implementation consists in reducing the number of simple bounds[9] examined in each iteration of the FM algorithm by representing with a single loop a set of loops that are related by the strip-mining transformation. Moreover, besides having a much lower complexity, our implementation also generates fewer redundant bounds. We will show this fact later in Section 4.6.

The remainder of this section is organized as follows: First, we present a theorem that holds in the SMT context and allows us to establish the complexity of the SMT implementation. Second, we show how the number of simple bounds examined in each iteration of the FM can be reduced. Third, we summarize the steps of the efficient implementation of SMT and give some implementation details and, finally, we present the SMT algorithm and illustrate how it works with an example.

4.4.1 Computing the Bounds of TI-loops

As stated in Section 4.2.2, the Fourier-Motzkin algorithm computes the exact loop bounds in the transformed space from innermost to outermost. Each iteration of FM performs two steps. In the second step, each of the lower simple bounds of the loop being solved is compared with each of the upper simple bounds. These comparisons generate inequalities that might become new simple bounds of the yet-to-be-processed loops and, in the worst case, the number of new simple bounds added grows quadratically in each iteration [129].

In a multilevel tiled code, the inner loops are always the loops that traverse the points within the tiles (the EL-loops) and the outer loops are the loops that step between tiles (the TI-loops). Thus, in our SMT algorithm, the bounds of the EL-loops are always computed before the bounds of the TI-loops. The following theorem holds when computing the loop bounds in the transformed space.

9.Recall that the loop bounds are max or min compositions of affine functions of the surrounding loop iteration variables and that we refer to each affine function as a *simple bound*.

Theorem 2

In SMT, the second step of the FM algorithm does not need to be performed when the loop being solved is a TI-loop.

Proof

In Appendix D, we demonstrate that, when computing the bounds of a TI-loop, all new simple bounds generated by the second step of the FM algorithm are redundant. Therefore, the second step of the FM algorithm does not need to be performed when the loop being solved is a TI-loop. ■

This theorem is very important because it allows us to demonstrate that the number of simple bounds does not increase quadratically when computing the bounds of TI-loops.

4.4.2 Examining fewer Simple Bounds

To compute the bounds of a certain loop, the FM algorithm examines, besides its own simple bounds, the simple bounds of the loops that are between its original position (before moving) and its final position (after moving). We note that it is not necessary to examine the bounds of other outer loops because they cannot have simple bounds that are affine functions of the loop being solved. In the example of Fig. 4.8, to compute the bounds of the innermost loop (loop `I`) in Fig. 4.8f, the FM algorithm examines the simple bounds of loops `I`, `JJ` and `J` in Fig. 4.8d. Note that the examined loops can be either *contiguous* TI-loops associated to the same EL-loop or *contiguous* TI -loops followed by their associated EL-loop.

In our implementation of SMT, we represent with a single loop (called C-loop; *Cluster loop*) all the contiguous TI-loops associated to the same EL-loop that must be examined by the FM algorithm. A C-loop traverses the same iteration space as the loops it is representing but uses a fewer number of simple bounds. Then, the idea of our implementation consists in stripping[10] the TI and EL-loops of the final code from their associated C-loops as late as possible, that is, just before their bounds have to be computed. Initially, all the loops in the original code are C-loops.

For example, assume that after strip-mining all loops of a loop nest `I-J-K` at all desired levels, we obtain the loop nest `III-I-JJ-J-KKK-KK-K`. The length of the loop index identifiers (`KKK-KK-K`) refers to the level of the tiles that they are traversing. Suppose that we want loop `I` to be in the innermost position in the final code. When computing its new bounds, the FM algorithm needs to examine all simple bounds of loops `I`, `JJ`, `J`, `KKK`, `KK` and `K`. By contrast, in the efficient implementation, we start with the loops in the original code `I-J-K`. These three loops are C-loops because they represent contiguous associated TI and EL-loops of the final code. Just before computing the bounds of EL-loop `I`

10.In this context, "to strip" means extracting the innermost TI (or EL)-loop from a C-loop.

in the final code, it is stripped from its associated C-loop, obtaining the loops II-I-J-K (loop II is now the new C-loop). Now, to compute the bounds of EL-loop I, the FM algorithm only examines the bounds of the loops I, J and K.

To implement this "stripping" process, the efficient SMT algorithm has to be able to strip from a C-loop the EL-loop and its associated TI-loops, one by one, from the innermost to the outermost level. Note that this order is just the reverse order of what strip-mining does. Thus, we need a backward transformation of strip-mining, that we will refer to as *strip-clustering*.

Strip-clustering

We define *strip-clustering* as a loop transformation that clusters a set of strips together. It decomposes a C-loop into two loops where the outer loop steps between clusters of inner strips and the inner loop steps between the initial points of the inner strips. The outer loop is a new C-loop and the inner loop is the TI or EL-loop being stripped. In strip-clustering, the boundaries of the tiles are parallel to the iteration space axes as in the strip-mining transformation.

The strip-clustering transformation applied to a C-loop with a step equal to one is defined by the expressions shown in Fig. 4.9, where II is the new C-loop, I is the EL-loop being stripped, B_{II} is the strip size of the new C-loop and $oft_{II} \in Z$ ($0 \leq oft_{II} < B_{II}$) is an offset that determines the origin of the first strip. Note that these expressions are exactly the same expressions as used in strip-mining.

$$\text{do I = L, U} \xrightarrow{\textbf{strip-clustering}} \begin{array}{l} \text{do II} = \lfloor (L - oft_{II}) / B_{II} \rfloor * B_{II} + oft_{II},\ U,\ B_{II} \\ \quad \text{do I} = \max(II, L),\ \min(II+B_{II}-1,\ U) \end{array}$$

Figure 4.9: Formula for strip-clustering a C-loop with a step equal to one.

The strip-clustering transformation applied to a C-loop with a step different from one is defined by the expressions in Fig. 4.10, where III is the new C-loop, II is the TI-loop being stripped, B_{III} is the strip size of the new C-loop, B_{II} is the strip size of the TI-loop ($B_{III} >> B_{II}$) and oft_{II}, $oft_{III} \in Z$ ($0 \leq oft_{II} < B_{II}$, $0 \leq oft_{III} < B_{III}$) are the offsets of each strip.

$$\begin{array}{l} \text{do II} = \lfloor (L - oft_{II}) / B_{II} \rfloor * B_{II} + oft_{II},\ U,\ B_{II} \\ \quad \text{do I} = \max(II, L),\ \min(II+B_{II}-1,\ U) \end{array}$$

$$\Big\downarrow \textbf{ Strip-clustering loop II}$$

$$\begin{array}{l} \text{do III} = \lfloor (L - oft_{III}) / B_{III} \rfloor * B_{III} + oft_{III},\ U,\ B_{III} \\ \quad \text{do II} = \lfloor (\max(III, L) - oft_{II}) / B_{II} \rfloor * B_{II} + oft_{II},\ \min(U,\ III+B_{III}-1),\ B_{II} \\ \quad\quad \text{do I} = \max(III, II, L),\ \min(III+B_{III}-1,\ II+B_{II}-1,\ U) \end{array}$$

Figure 4.10: Formula for strip-clustering a C-loop with a step different from one.

Two different phases can be distinguished when strip-clustering is applied to a C-loop with a step different from one:

- **Creating phase**: This phase consists in creating the new C-loop and the TI-loop being stripped. The bounds of the C and TI-loops are computed directly using the expressions of Fig. 4.10.

- **Broadcasting phase**: This phase consists in modifying the simple bounds of all earlier stripped TI and EL-loops (generated by strip-clusterings applied previously). In the general case, the TI-loop `II` being stripped (see Fig. 4.10) can iterate over some points outside the tiles determined by the new C-loop `III` (that is, `III` $\leq$ `II` and `II`+B_{II} -1 $\leq$ `III`+B_{III}-1 does not hold in the general case). Therefore, two new simple bounds (the lower simple bound `III` and the upper simple bound `III`+B_{III} -1) have to be added to the bounds of all previous stripped TI and EL-loops (loop `I` in Fig. 4.10). Note that adding these simple bounds is equivalent to *substituting iteration variable* `II` *by* max(`III`,`II`) *in the lower bound and by* min(`II`+B_{II}-1,`III`+B_{III}-1)-B_{II}+1 *in the upper bound* of all previous stripped TI and EL-loops.

Note also that when strip-clustering is applied to a C-loop with a step equal to one (Fig. 4.9) only the *creating phase* is performed since there are no inner TI or EL-loops previously stripped.

Figure 4.11 shows graphically how the bounds of previously stripped TI and EL-loops have to be modified in the *broadcasting phase*. Figure 4.11a shows how the iteration space is traversed before applying the strip-clustering transformation to a C-loop `II` with a step different from one and Fig. 4.11b shows the iteration space after applying the transformation. The points on each line indicate the values of the loop index variable and the shadowed rectangles indicate the values of a loop index variable for fixed values of outer loop indices. The strip sizes at each level are B_{III}=17 and B_{II}=5 and, for simplicity, we assume null offsets (oft_{III}=oft_{II}=0).

Computing the Minimum Convex Space after Strip-Clustering

Finally, we want to clarify that the FM algorithm is executed using the bounds of the MCS of the NCBIS that are directly obtained from the NCBIS. Then, the new bounds obtained after the execution of FM are the bounds of the MCS of the BTIS, that must be corrected using the vector oft^P and the matrix H^P to obtain the exact bounds of the BTIS.

Recall from Definition 2 (Section 4.3.3) that a loop I_k^{NCBIS} in NCBIS can be written, in the general case, into the following form:

$$\text{do } I_k^{NCBIS} = \left\lfloor \left(L_k^{NCBIS} - oft_k \right) / B_k \right\rfloor \cdot B_k + oft_k, U_k^{NCBIS}, B_k$$

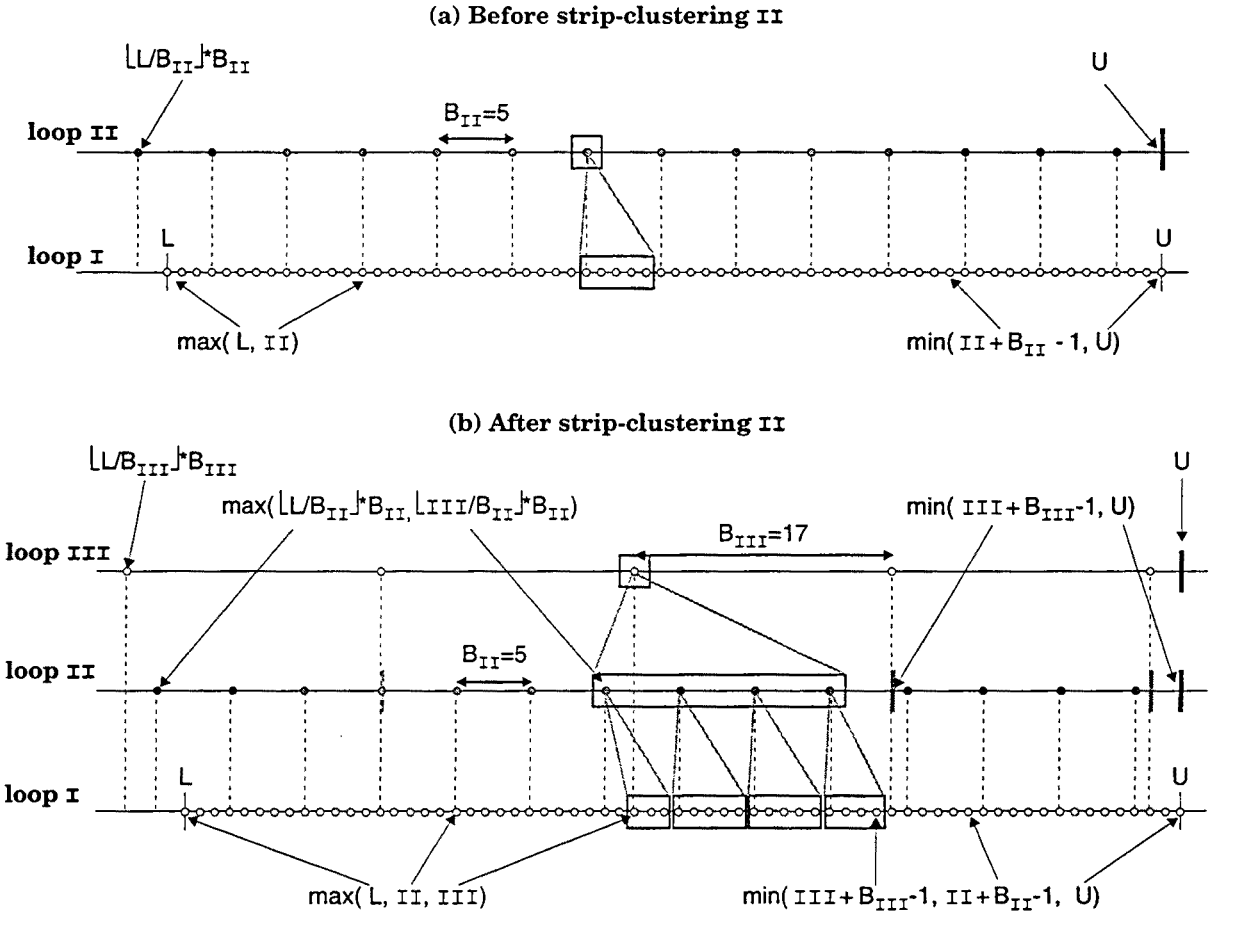

Figure 4.11: (a) Iteration space before strip-clustering a C-loop II with a step different from one. (b) Iteration space after strip-clustering a C-loop II with a step different from one.

and recall from Corollary 2 that the bounds of the MCS of NCBIS along dimension I_k^{NCBIS} can be directly obtained from the bounds of NCBIS as follows:

$$L_k^{MCS} = L_k^{NCBIS} - B_k + 1$$
$$U_k^{MCS} = U_k^{NCBIS}$$

Therefore, in our implementation of SMT, we will directly work with the bounds of the MCS of the NCBIS. This means that when we apply strip-clustering to a loop, we do not directly use the bounds obtained using the expressions in Figures 4.9 and 4.10. Instead, we use the bounds of the MCS.

4.4.3 Steps of the Efficient SMT Implementation

Our implementation of SMT will generate the EL and TI loops of the final multilevel tiled code, one by one, from the innermost to the outermost one, by integrating the strip-clustering transformation with the loop bounds computation.

Firstly, the loops of the original code are the initial C-loops. In each iteration, the SMT algorithm performs two different actions:

- First, it applies the strip-clustering transformation to a C-loop, generating the new C-loop and the TI or EL-loop whose bounds are going to be computed.
- Second, the exact bounds of this just stripped TI or EL-loop are computed performing one iteration of the FM algorithm. If the just stripped loop is an EL-loop both steps of the FM algorithm are performed and if it is a TI-loop, only the first step is performed (Theorem 2).

As an example, suppose that we have a 3-deep loop nest (I-J-K) and we want to obtain the multilevel tiled code (KKK-III-JJ-II-K-J-I). Figure 4.12 shows the order in which strip-clustering and the computation of the bounds are performed by our implementation of SMT. The loop list written in the rows labelled *"strip-clustering"* indicates which C-loop (in bold) and which TI or EL-loop (in bold+oblique) appear after strip-clustering. In rows labelled *"compute bounds"*, we indicate that the TI or EL-loop just stripped is moved to the innermost position and that its bounds are being solved performing one iteration of FM.

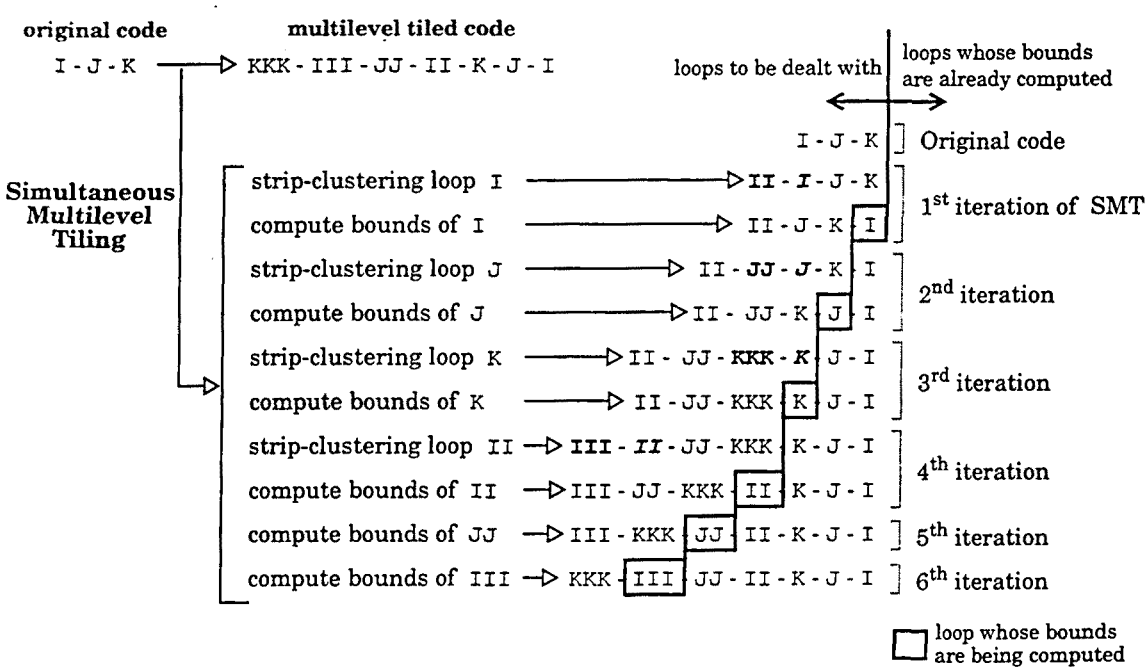

Figure 4.12: Example of how the implementation of SMT works.

In the three first iterations of SMT only the *creating phase* of strip-clustering is performed, because the C-loops have steps equal to one. In the fourth iteration, strip-clustering is applied to a C-loop that has a step different from one (loop II), and therefore, both the *creating and the broadcasting phases* are performed.

In the broadcasting phase we have to take into account that there could be other loops between the just stripped TI-loop (II) and its associated EL-loop (I). These loops can have simple bounds that are affine functions of the just stripped TI-loop and, therefore, their bounds must also be modified in the broadcasting phase. In Fig. 4.12, 4^{th} iteration of SMT, the bounds of loops JJ, KKK, K, J and I must be modified.

Theorem 3

Let III and II be the loop index variables of a C-loop and the just stripped TI-loop and let B_{III} and B_{II} be the strip sizes of III and II, respectively. The bounds of the loops between the just stripped TI-loop II and its associated EL-loop (including the EL-loop) must be modified in the broadcasting phase in the following manner: *iteration variable* II *has to be substituted by* max(III, II) *if it appears in a lower (upper) simple bound and the coefficient that multiplies* II *is positive (negative) and by* min(II+B_{II}-1, III+B_{III}-1) -B_{II}+1 *if it appears in an upper (lower) simple bound and the coefficient is positive (negative).*

Proof

Lets consider the following loop nest:

```
do I = L, U
  do J = α1 · I + θ1 , α2 · I + θ2
enddo
```

where $\alpha_1, \alpha_2, \theta_1, \theta_2$, L and U are integer constants or program parameters (variables unchanged within the loops). Let assume that $\alpha_1, \alpha_2 > 0$ and $\alpha_1 \leq \alpha_2$.

Let's perform multilevel tiling using our efficient implementation of SMT to achieve the tiled code III-II-J-I (from outermost to innermost). First, strip-clustering is applied to loop I, obtaining the following code:

```
do II = ⌊(L - oft_II) / B_II⌋ * B_II + oft_II, U, B_II
  do I = max(II, L), min(II+B_II-1, U)
    do J = α1 · I + θ1 , α2 · I + θ2
enddo
```
(1)

Let F^L and F^U be the lower and upper simple bounds added by the strip-clustering transformation to EL-loop I. In loop nest (1), F^L = II and F^U = II+B_{II}-1.

Then, loop nest (1) can be rewritten as follows:

$$\begin{array}{l} \text{do II} = \lfloor (L - oft_{II}) / B_{II} \rfloor * B_{II} + oft_{II},\ U,\ B_{II} \\ \quad \text{do I} = \max(F^L, L),\ \min(F^U, U) \\ \quad\quad \text{do J} = \alpha_1 \cdot I + \theta_1,\ \alpha_2 \cdot I + \theta_2 \\ \text{enddo} \end{array} \qquad (2)$$

Second loops I and J in (2) are permuted performing one iteration of the FM algorithm. After the loop permutation, the following code is obtained:

$$\begin{array}{l} \text{do II} = \lfloor (L - oft_{II}) / B_{II} \rfloor * B_{II} + oft_{II},\ U,\ B_{II} \\ \quad \text{do J} = \max(\alpha_1 \cdot F^L + \theta_1,\ \alpha_1 \cdot L + \theta_1,\ \left\lfloor \frac{\alpha_2 \cdot \theta_1 - \alpha_1 \cdot \theta_2}{\alpha_2 - \alpha_1} \right\rfloor),\ \min(\alpha_2 \cdot F^U + \theta_2,\ \alpha_2 \cdot U + \theta_2) \\ \quad\quad \text{do I} = \max(F^L, L, \lceil (J - \theta_2) / \alpha_2 \rceil),\ \min(F^U, U, \lfloor (J - \theta_1) / \alpha_1 \rfloor) \\ \text{enddo} \end{array}$$

And third, strip-clustering is applied to loop II. The strip-clustering transformation applied to a loop with a step different from one consists of two phases: the creating phase and the broadcasting phase. After the creating phase of strip-clustering, the following loop nest is obtained (where F^L = II and F^U = II+B_{II}-1):

$$\begin{array}{l} \text{do III} = \lfloor (L - oft_{III}) / B_{III} \rfloor * B_{III} + oft_{III},\ U,\ B_{III} \\ \quad \text{do II} = \lfloor (\max(III, L) - oft_{II}) / B_{II} \rfloor * B_{II} + oft_{II},\ \min(U, III+B_{III}-1),\ B_{II} \\ \quad\quad \text{do J} = \max(\alpha_1 \cdot F^L + \theta_1,\ \alpha_1 \cdot L + \theta_1,\ \left\lfloor \frac{\alpha_2 \cdot \theta_1 - \alpha_1 \cdot \theta_2}{\alpha_2 - \alpha_1} \right\rfloor),\ \min(\alpha_2 \cdot F^U + \theta_2,\ \alpha_2 \cdot U + \theta_2) \\ \quad\quad\quad \text{do I} = \max(F^L, L, \lceil (J - \theta_2) / \alpha_2 \rceil),\ \min(F^U, U, \lfloor (J - \theta_1) / \alpha_1 \rfloor) \\ \text{enddo} \end{array} \qquad (3)$$

If strip-clustering were applied to loop II before performing the loop permutation, that is, in loop nest (2), F^L and F^U would be F^L = max (III, II) and F^U = min(III+B_{III}-1, II+B_{II}-1) by definition of strip-clustering and, the bounds of loops I and J after the loop permutation would be the same as in loop nest (3) but with the new values of F^L and F^U.

Therefore, in the broadcasting phase of strip-clustering, the bounds of the loops between the just stripped TI-loop II and its associated EL-loop I (including the EL-loop) must be modified as follows:

- F^L = II must be substituted by max (III, II) in the lower bounds and
- F^U = II+B_{II}-1 must be substituted by min(III+B_{III}-1, II+B_{II}-1) in the upper bounds

that is, iteration variable II must be substituted by max(III, II) in the lower bounds and by min(II+B_{II}-1, III+B_{III}-1)-B_{II}+1 in the upper bounds.

To demonstrate the substitution in the presence of negative coefficient, assume $\alpha_1, \alpha_2 < 0$ and perform in a similar way. ■

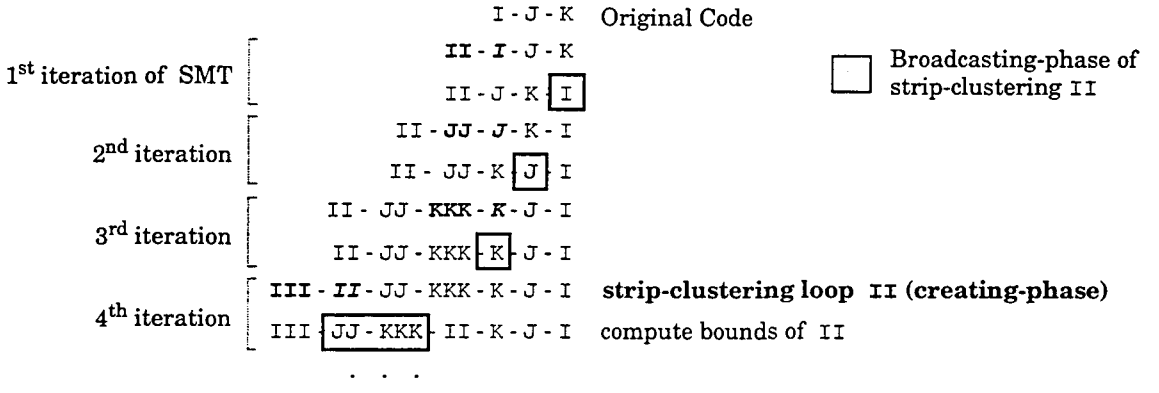

Figure 4.13: Modifications of loop bounds due to the broadcasting phase of strip-clustering loop II.

Notice that, in the broadcasting phase of strip-clustering, for each simple bound that is an affine function of the just stripped TI-loop, one and only one simple bound that is an affine function of the new C-loop is added.

Implementation Details

We have seen that when strip-clustering is applied to a particular loop for stripping a TI-loop, the bounds of the loops between the just stripped TI-loop and its associated EL-loop (including the EL-loop) must be modified (broadcasting phase). In each iteration of SMT we perform one iteration of the FM algorithm and, in each iteration of the FM algorithm, the bounds of the yet-to-be processed loops are examined. In this section, we present an efficient implementation of the broadcasting phase that essentially consists in joining together the broadcasting phase with the bounds inspection performed in the FM algorithm.

To illustrate the implementation we will use the example shown in Figure 4.13 (same example shown in Fig. 4.12). Figure 4.13 shows, with a shadowed rectangle, when the modification of bounds due to the broadcasting phase of strip-clustering loop II (4^{th} iteration of SMT) is done for each loop.

When strip-clustering is applied to a C-loop having a step different from one, the loops between the just stripped TI-loop and its associated EL-loop can be divided into two groups: a) loops that have already been processed and, therefore, their bounds have already been computed and b) loops that are not yet processed and, therefore, their bounds are not yet computed. In Fig. 4.13 (4^{th} iteration of SMT), when strip-clustering is applied to C-loop II, loops K, J and I have already been processed, while loops JJ and KKK are not yet processed. Then, the implementation of the broadcasting phase consists in performing the modification of bounds for the inner loops in the following manner:

- **The modification of bounds for loops that have already been processed** could be done just when their bounds were computed, because the number of levels and the tile sizes that each space dimension is partitioned are known before starting the execution of SMT. In Fig. 4.13, the modification of bounds for loops I, J and K due to the *broadcasting*

phase of strip-clustering II is performed just after the bounds of I, J and K have been computed in the 1st, 2nd and 3rd iteration of SMT, respectively. Note that the new bounds added by the broadcasting phase will never be involved in later iterations of the SMT algorithm because these loops have already been processed.

• **The modification of bounds for loops that are not yet processed** can be done at the same time as the bounds of the just stripped TI-loop are computed. To compute the bounds of the just stripped TI-loop, the first step of one iteration of the FM algorithm is performed. In this step, all simple bounds of the yet-to-be-processed loops are examined and all simple bounds that are affine functions of the TI-loop being solved become simple bounds of the TI-loop. Then, for each of these simple bounds that are affine functions of the stripped TI-loop, we can add (to the corresponding loop) the new loop bound due to the broadcasting phase. These extra new bounds do not need to be examined by the FM algorithm because they are not affine functions of the stripped TI-loop (they are affine functions of the associated C-loop). Therefore, the number of bounds examined in the current execution of the FM algorithm does not increase. In Fig. 4.13, the modification of bounds for loops JJ and KKK due to the broadcasting phase of strip-clustering II is done at the same time as the bounds of the stripped TI-loop II are computed.

4.4.4 SMT Algorithm

Summarizing, our implementation of the SMT algorithm consists in integrating the strip-clustering transformation inside the FM algorithm to compute the bounds of the MCS of the BTIS (the multilevel tiled iteration space). In each iteration, the SMT algorithm executes the following steps:

• First, it performs the *creating phase* of the strip-clustering transformation, generating the new C-loop and the TI or EL-loop whose bounds are going to be computed.

• Second, it computes the bounds of the just stripped TI or EL-loop performing one iteration of FM algorithm. If the loop is an EL-loop, the SMT algorithm performs both steps of the FM algorithm. However, if the loop is a TI-loop, it only performs the first step of FM (Theorem 2) and, at the same time, it does the modification of bounds for the loops whose bounds are not yet computed (due to the *broadcasting phase* of the just applied strip-clustering).

• Third, it does the modification of bounds for the loop being solved (due to the *broadcasting phases* of strip-clusterings that will be later applied).

Finally, at the end of the SMT process, the bounds of the MCS of BTIS are corrected using vector oft^P and matrix H^P, as shown in Section 4.3.3 (Theorem 1), to obtain the exact bounds of BTIS.

```
Algorithm
INPUT: TL  /* ordered list (from inner to outermost) with the names of the loop indices in the tiled loop nest. TL[0] is
           the innermost loop */
       OL  /* ordered list (from inner to outermost) of the loops in the original code */
       m   /* total number of loops in the multilevel tiled loop nest */
       n   /* total number of loops in the original loop nest */
       H^P /* mxm diagonal matrix where the k^th diagonal element is the step of the k^th loop in the tiled loop nest */
       oft^P /* m-dimensional column vector where the k^th element is the offset of the k^th loop in the tiled loop nest */
OUTPUT: FL  /* loop nest after Multilevel Tiling */
  FL = OL;
  s = n;  /* number of loops in FL */
  r = 0;  /*  number of loops that have already been processed */
  for (i = 0; i < m-1; i++)
  { j = position of TL[ i ] in FL;
    AL = Extract ( j , FL);      /* extract from FL the loop in position j. AL is the active loop */
    /* apply creating phase of strip-clustering to AL */
    if (AL.levels > 0)
    { (C-loop, AL) = apply creating phase of strip-clustering to AL; /* AL is now the stripped TI or EL-loop */
      C-loop.levels = C-loop.levels-1;
      insert (C-loop , j , FL);      /* insert the C-loop in position j of FL */
      str_cl = TRUE;      /* indicates that strip-clustering has been performed in this iteration */
    }
    else str_cl = FALSE;
    /* compute bounds of AL and modify bounds of not-yet processed loops due to the broadcasting phase of the
       just applied strip-clustering */
    if (AL is an EL-loop)
    { solve AL from the simple bounds of loops in FL in positions r to j-1;                    /* 1st step of FM */
      compare lower and upper bounds of AL and add the new bounds to loops in FL;   /* 2nd step of FM */
    }
    if (AL is a TI-loop)
    { if (str_cl==TRUE)
        /* 1st step of FM+modification of bounds due to broadcasting phase of the just applied strip-clustering */
        solve AL from the simple bounds of loops in FL in positions r to j-1 and, at the same time, add to these
        loops in positions r to j-1 the simple bounds due to the broadcasting phase of just applied strip-clustering;
      else /* perform only the 1st step of FM */
        solve AL from the simple bounds of loops in FL in positions r to j-1;
    }
    /* modify bounds of the loop being solved due to later applied strip-clusterings*/
    add simple bounds to AL due to broadcasting phases of later applied strip-clusterings;
    insert (AL , r , FL);      /* insert AL in the innermost position */
    if (str_cl == TRUE) s=s+1;
    r=r+1;
  }
  correct the bounds of the loops in FL using H^P and oft^P;
endAlgorithm
```

Figure 4.14: Code of the efficient SMT algorithm.

The complete SMT algorithm is shown in Figure 4.14. List OL contains all the information related to the n loops in the original loop nest (simple bounds, name of the iteration variable, number of levels being exploited by this loop (field `levels`), etc.). List TL contains the names of the m iteration variables in the resulting tiled loop nest, ordered from innermost to outermost. We want to obtain the ordered list FL that contains all loops in the final tiled code with all the simple bounds computed. FL is initialized to OL and at the end of the algorithm it will contain the multilevel tiled code. Other variables used in the algorithm are: a) variable s that indicates the number of loops in list FL, b) variable r that indicates the number of loops that have already been processed and c) variable str_cl that indicates, in each iteration, if strip-clustering has been performed or not.

List TL gives the order in which the loops are processed. Thus, for each of the first m-1 iteration variables in TL[11], the algorithm begins by finding the position j in FL where the associated C-loop of the TI or EL-loop we want to deal with is. This C-loop is then assigned to AL (Active Loop) and removed from FL.

Next, the *creating phase* of strip-clustering is applied to AL, obtaining the new C-loop and the TI or EL-loop whose bounds are going to be computed; the new C-loop is inserted in list FL in position j and the TI or EL-loop is assigned to AL. Thereafter, the SMT algorithm computes the bounds of AL (the just stripped TI or EL-loop). If AL is an EL-loop, both steps of the FM algorithm are performed. If AL is a TI-loop, only the first step of FM is performed and, at the same time, the modification of bounds due to the *broadcasting phase* of the just applied strip-clustering is done.

Then, the SMT algorithm performs the modification of bounds for loop AL (due to the *broadcasting phases* of strip-clusterings that will be later applied). Finally, loop AL is inserted in FL in position r (the innermost position).

At the end of the SMT process, the bounds of the loops in FL are corrected to obtain the exact bounds of the multilevel tiled code.

Execution Example

To exemplify our implementation of SMT, we will use the same example as in Section 4.3.5 (Fig. 4.8a). Again, we assume that tiling has to be performed at two levels (at the innermost level, dimensions `I` and `J` have to be tiled and, at the outermost level, only dimension `I` has to be tiled) and we use all offsets equal to 1 in the strip-clustering transformation. The desired loop order in the multilevel tiled code is `III-JJ-II-J-I`, from outermost to innermost.

We will show the bounds of the loops at intermediate iterations of the SMT algorithm. Recall that our SMT implementation works with the MCS defined by the bounds of the loops. For clarity, we present this information in loop nest form. However, we note that these loop nests are not

11.The bounds of the outermost loop are obtained directly after the SMT algorithm has finished.

1st iteration of SMT:

(a) Original Code

```
do I = 1, N
  do J = I, N
    loop body
enddo
```

(b) Strip-clustering I

```
do II = { 2-B_II, N }
  do I = { max(II, 1), min(II+B_II-1, N) }
    do J = { I, N }
```

(c) Computing bounds of I

```
do II = { 2-B_II, N }
  do J = { max(II, 1), N }
    do I = { max(II, 1), min(II+B_II-1, N, J) }
```

(d) Modification of bounds for loop I due to broadcasting phase of II

```
do II = { 2-B_II, N }
  do J = { max(II, 1), N }
    do I = { max(II, III, 1), min(II+B_II-1, III+B_III-1, N, J) }
```

2nd iteration of SMT:

(e) Strip-clustering J

```
do II = { 2-B_II, N }
  do JJ = { max (II, 1) - B_JJ+1, N }
    do J = { max(II, 1, JJ), min(N, JJ+B_JJ-1) }
      do I = { max(II, III, 1),
               min(II+B_II-1, III+B_III-1, N, J) }
```

(f) Modification of bounds for loop J due to broadcasting phase of II

```
do II = { 2-B_II, N }
  do JJ = { max (II, 1) - B_JJ+1, N }
    do J = { max(II, III, 1, JJ), min(N, JJ+B_JJ-1) }
      do I = { max(II, III, 1),
               min(II+B_II-1, III+B_III-1, N, J) }
```

3rd iteration of SMT:

(g) Strip-clustering II

```
do III = { 2-B_III, N }
  do II = { max(III, 1) -B_II+1, min(III+B_III-1, N) }
    do JJ = { max (II, 1) - B_JJ+1, N }
      do J = { max(II, III, 1, JJ), min(N, JJ+B_JJ-1) }
        do I = { max(II, III, 1),
                 min(II+B_II-1, III+B_III-1, N, J) }
```

(h) Computing bounds of II (only 1st step of FM) and modification of bounds due to broadcasting phase of II

```
do III = { 2-B_III, N }
  do JJ = { max (III, 1) - B_JJ+1, N }
    do II = { max(III, 1) -B_II+1,
              min(III+B_III-1, N, JJ+B_JJ-1)}
      do J = { max(II, III, 1, JJ), min(N, JJ+B_JJ-1) }
        do I = { max(II, III, 1),
                 min(II+B_II-1, III+B_III-1, N, J) }
```

(i) Correcting bounds. Multilevel Tiled Code

```
do III =1, N, B_III
  do JJ = ⌊(max(III-1, 0)/ B_JJ⌋*B_JJ+1, N, B_JJ
    do II = ⌊max(III-1, 0)/ B_II⌋*B_II+1, min(III+B_III-1, N, JJ+B_JJ-1), B_II
      do J = max(II, III, 1, JJ), min(N, JJ+B_JJ-1)
        do I = max(II, III, 1), min(II+B_II-1, III+B_III-1, N, J)
          loop body
enddo
```

Figure 4.15: (a) Original code. (b) Loop nest after strip-clustering loop I. The braces on the loop bounds indicate that they are representing the MCS. (c) Loop nest after performing one iteration of the FM algorithm to compute the bounds of I. (d) Loop nest after modifying the bounds of loop I due to later applied strip-clusterings. (e) Loop nest after strip-clustering loop J. (f) Loop nest after modifying the bounds of J due to later applied strip-clusterings. (g) Loop nest after strip-clustering loop II. (h) Loop nest after performing the first step of the FM algorithm to compute the bounds of II and after modifying the bounds of JJ due to the broadcasting phase of the just applied strip-clustering. (i) Loop nest after correcting the bounds of the MCS of the BTIS. This is also the final multilevel tiled code.

semantically correct until the SMT algorithm finishes (we use braces on the loop bounds to indicate that they are representing the MCS). The loops whose bounds have already been computed are the innermost loops in the nest while the not-yet-processed loops are the outermost loops.

Initially, loops I and J in the original code are the C-loops (Fig. 4.15a). In the first iteration of SMT, strip-clustering is applied to loop I (the innermost loop in the final code), obtaining the bounds shown in Fig. 4.15b. Then, the bounds of loop I are computed performing both steps of the FM algorithm (Fig. 4.15c) and, finally, the bounds of loop I are modified due to the broadcasting phase of the strip-clustering transformation that will be later applied (in the 3rd iteration of SMT) to loop II (Fig. 4.15d).

In the second iteration of SMT, strip-clustering is applied to loop J (the next innermost loop in the final code), obtaining the bounds shown in Fig. 4.15e. Then, the bounds of loop J are modified due to the broadcasting phase of the strip-clustering transformation that will be later applied to loop II (Fig. 4.15f). Note that in this iteration it is not necessary to perform one iteration of the FM algorithm to compute the exact bounds.

In the third iteration of SMT, strip-clustering is applied to loop II (only the creating phase), obtaining the bounds shown in Fig. 4.15g. Then, the bounds of loop II are computed performing only the first step of the FM (loop II is a TI-loop). Moreover, at the same time as the bounds are computed, we add to loop JJ the new bounds due to the broadcasting phase of the just applied strip-clustering. (Fig. 4.15h). At this point we already have the bounds of the Minimum Convex Space of BTIS.

Finally, the bounds must be corrected as shown in Section 4.3 to obtain the exact bounds of the final multilevel tiled code. Figure 4.15i shows the bounds after they have been corrected.

4.5 COMPLEXITY OF THE EFFICIENT SMT IMPLEMENTATION

In this section we analyze the complexity of the efficient SMT algorithm (Figure 4.14). We will first present two lemmas that establish the complexity of computing the bounds of the EL-loops and of the TI-loops. Then, we enounce a theorem that yields the complexity of the SMT algorithm.

Let n and q be the number of loops and simple bounds in the original loop nest, respectively, and let m be the number of loops in the resulting multilevel tiled code. Our implementation of SMT computes the bounds of the EL and TI loops of the final multilevel tiled code, one by one, from the innermost to the outermost one. Thus, the bounds of the n EL-loops are always computed before the bounds of the m-n TI-loops. Let's first look at the complexity of computing the bounds of the EL-loops.

Lemma 2

The complexity of computing the bounds of the n EL-loops is the same complexity as performing a loop permutation in the original code, that is:

$$C_{EL} = O\left(\left(\frac{q}{2}\right)^{2^n-1}\right)$$

Proof

To compute the bounds of an EL-loop, the SMT algorithm performs the following steps: first, the creating phase of strip-clustering is applied to a C-loop for stripping the EL-loop whose bounds are going to be computed; second, one iteration (first and second step) of the FM algorithm is executed; and third, the bounds of the EL-loop being solved are modified due to the *broadcasting phases* of strip-clusterings that will be later applied

In the worst case, the first loop iteration variable to be solved is involved in the q simple bounds of the original loop nest. Therefore, the number of simple bounds involved the first time that one iteration of the FM algorithm is executed is $q + 2$ (q simple bounds of the original code and 2 simple bounds introduced by the creating phase of previous applied strip-clustering). We can round off $q + 2$ to q (that is, $q + 2 \approx q$), because $q \gg 2$.

Then, after performing the second step of the FM algorithm the number of simple bounds of the next loop iteration variable to be solved can be, in the worst case, $(q/2)^2$. Recall that the second step of FM makes the number of simple bounds of the yet-to-be processed loops to increase quadratically.

Finally, after executing the FM algorithm, the bounds of the EL-loop being solved are modified due to the *broadcasting phases* of strip-clusterings that will be later applied. This step, however, does not introduce new bounds to the yet-to-be processed loops.

Therefore, since all q original simple bounds could potentially involve the n original iteration variables, the worst-case complexity of computing the bounds of the n EL-loops is:

$$C_{EL} = O\left(\left(\frac{q}{2}\right)^{2^n-1}\right)$$

This is also the complexity of performing a loop permutation in the original code.■

Let's now see the complexity of computing the bounds of the TI-loops.

Lemma 3

The worst-case complexity of computing the bounds of the m-n TI-loops is:

$$C_{TI} = O\left((m-n-1) \cdot \left(\frac{q}{2}\right)^{2^{n-1}} \right)$$

Proof

From Theorem 2, we have that the second step of the FM algorithm does not need to be performed when computing the bounds of a TI-loop. Then, to compute the bounds of a TI-loop, the SMT algorithm performs the following steps: first, the creating phase of strip-clustering is applied to a C-loop for stripping the TI-loop whose bounds are going to be computed; second, the first step of the FM algorithm is executed and, at the same time, the modification of bounds for the yet-to-be-processed loops (due to the *broadcasting phase* of the just applied strip-clustering) is done; and third, the bounds of the TI-loop being solved are modified due to the *broadcasting phases* of strip-clusterings that will be later applied

After computing the bounds of the n innermost EL-loops, the outer C-loops together can potentially have q_1 simple bounds (see Fig. 4.16), where:

$$q_1 = q + \left(\frac{q}{2}\right)^2 + \left(\frac{q^2}{8}\right)^2 + \ldots + \frac{q^{2^{n-1}}}{2^{2^n-2}} \approx \left(\frac{q}{2}\right)^{2^{n-1}}$$

In the worst case, the first TI-loop to be solved is involved in the q_1 simple bounds. Therefore, the number of simple bounds involved in the first step of the FM algorithm is $q_1 + 2$ (the q_1 simple bounds and 2 simple bounds introduced by the creating phase of strip-clustering). Again, we can round off $q_1 + 2 \approx q_1$.

At the same time as the first step of FM is executed, the modification of bounds for the yet-to-be-processed loops (due to the *broadcasting phase* of the just applied strip-clustering) is done. In this step, we introduce a new simple bound for each simple bound that is an affine function of the TI-loop being solved (Theorem 3). These extra new bounds do not need to be examined by the FM algorithm because they are not affine functions of the TI-loop (they are affine functions of the associated C-loop). Therefore, the modification of bounds does not increase the number of bounds examined in the current execution of the FM algorithm.

Finally, the bounds of the TI-loop being solved are modified due to the *broadcasting phases* of strip-clusterings that will be later applied. This step, however, does not introduce new bounds to the yet-to-be processed loops.

After computing the bounds of a TI-loop, the outer C-loops together will still have q_1 simple bounds because (1) for each simple bound that was an affine function of the TI-loop we have introduced one, and only one, new simple bound that is an affine function of its associated C-loop, (2) all simple bounds that were affine functions of the TI-loop become simple bounds of the TI-loop and (3) the second step of the FM algorithm is not performed.

Then, for each TI-loop to be solved, there are always q_1 bounds involved in the execution of the FM algorithm. There are *m-n* TI-loops in the multilevel tiled code. However, only *m-n-1* TI-loops are solved by the SMT algorithm. Recall that the bounds of the outermost TI-loop are directly obtained after the SMT algorithm has finished (Section 4.4.4). Thus, the worst-case complexity of computing the bounds of the TI-loops is:

$$C_{TI} = O((m-n-1)\cdot q_1) = O\left((m-n-1)\cdot\left(\frac{q}{2}\right)^{2^{n-1}}\right) \blacksquare$$

Theorem 4

The worst-case complexity of SMT is:

$$C_{SMT} = O\left((m-n)\cdot\left(\frac{q}{2}\right)^{2^{n-1}}\right)$$

Proof

Directly from Lemma 2 and Lemma 3. The complexity of SMT is the sum of the complexity of computing the EL-loops and the complexity of computing the TI-loops. Therefore,

$$C_{SMT} = C_{EL} + C_{TI} = O\left((m-n)\cdot\left(\frac{q}{2}\right)^{2^{n-1}}\right) \blacksquare$$

Notice that the complexity of SMT depends linearly on the number of TI-loops in the final code, rather than doubly exponentially, and it depends doubly exponentially only on the number of loops in the original code (before tiling). Thus, the complexity of SMT is proportional to the complexity of performing a loop permutation in the original loop nest.

Figure 4.16 shows, using the same example as in Figure 4.12, how the total number of simple bounds increases in each iteration of SMT (worst case). We give for each iteration the maximum number of simple bounds in different sets of loops (those enclosed in the arrows).

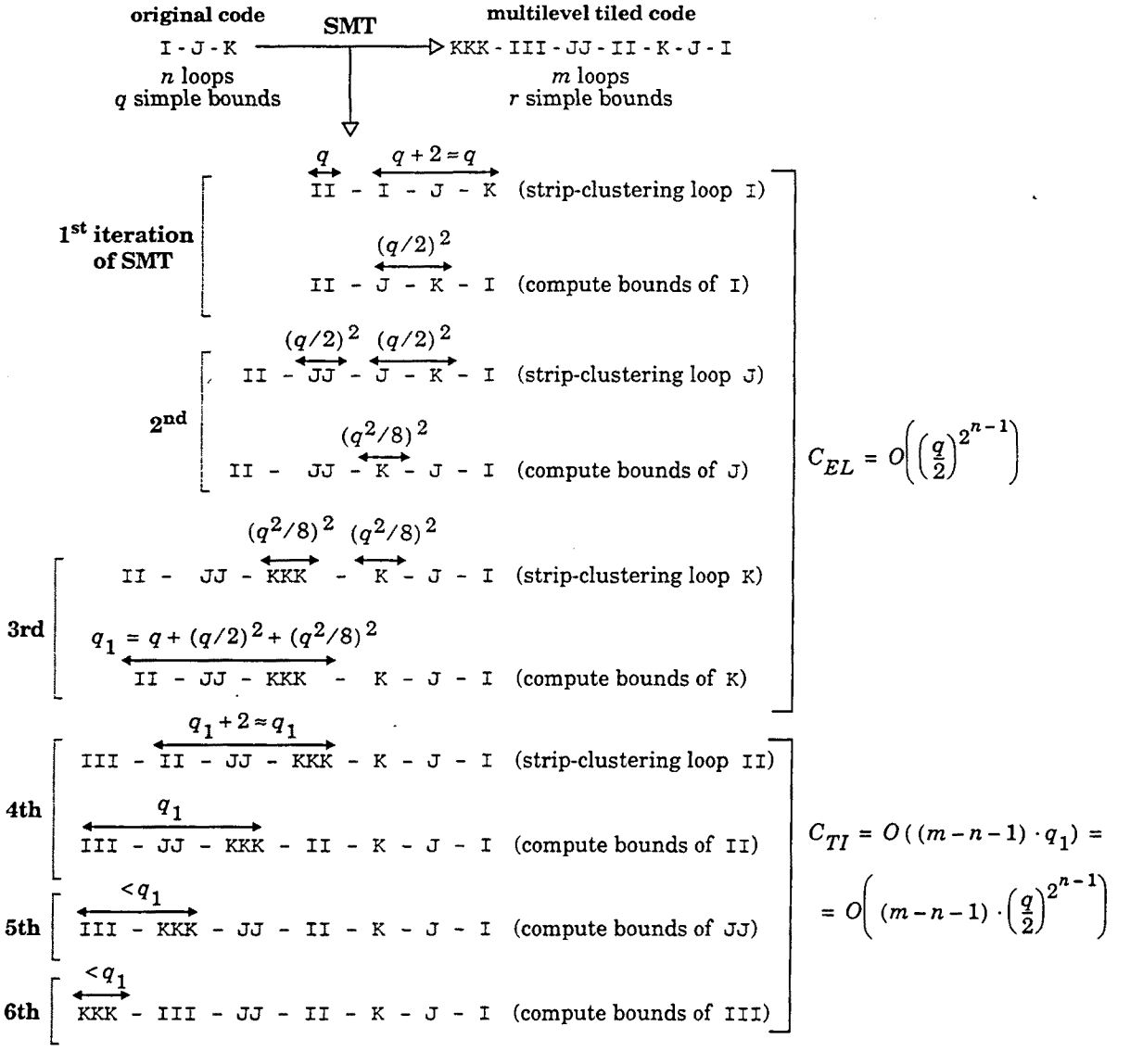

Figure 4.16: Complexity of the efficient implementation of SMT.

4.6 REDUNDANT BOUNDS

The Fourier-Motzkin algorithm used to compute the exact loop bounds can generate redundant bounds in the transformed loop nest [5][17]. The presence of redundant bounds in the transformed tiled code can be negative if index set splitting (ISS) is used after tiling to exploit the register level [60], because the number of times that ISS has to be performed and the amount of code generated both depend polynomially on the number of bounds the innermost loops have after tiling.

Every time ISS is performed, a loop nest is duplicated. If the number of generated loop nests increases excessively, the compiler might waste a lot of time performing the instruction scheduling and the register allocation of loop nests that will be never executed. Thus, the elimination of redundant bounds, at least in the innermost loops, is important to generate efficient code and to reduce compile-time when the register level is being exploited.

Futhermore, another advantage of eliminating redundant bounds is to avoid increasing program execution time. If the compiler generates excessive redundant bounds, a fraction of a program's execution time can be wasted in evaluating useless bounds (redundant bounds or bounds of loops that will end up in empty iterations[12]). This fraction of time is insignificant if tiling is applied to rectangular iteration spaces and/or for one or two levels of the memory hierarchy. However, it can become significant if tiling is applied to non-rectangular iteration spaces and for several levels of the memory hierarchy.

In this section we show how our implementation of Simultaneous Multilevel Tiling generates fewer redundant bounds than a conventional implementation. Moreover, we also show how our implementation allows removing the remaining redundant bounds in the innermost loops at a much lower cost than traditional implementations.

We distinguish two kinds of redundant simple bounds: trivial and non-trivial. A simple bound of a particular loop `I` is a *trivial* redundant simple bound if it can be deduced that it is redundant by only looking at the other simple bounds of loop `I`. These bounds can be eliminated as soon as they appear in an easy way [5]. A simple bound of a particular loop `I` is a *non-trivial* redundant simple bound if it is necessary to look at the simple bounds of outer loops to deduce that it is redundant. As an example, consider the code of Fig. 4.17a. Figure 4.17b shows the loop nest after applying a loop permutation transformation. The exact loop bounds of the transformed code have been computed using the Fourier-Motzkin algorithm. The upper simple bound `N` of loops `J` and `K` is a trivial redundant bound. However, the lower simple bounds `1` and `J` of loop `I` are non-trivial redundant bounds because it is necessary to know the bounds of `J` and `K` to deduce that they are redundant.

12.The FM algorithm computes exact bounds and therefore, the tiled loop nest never executes an empty iteration. However, if it contains redundant bounds and ISS is used after tiling, the final code can contain loop nests executing empty iterations.

(a) Original loop nest

```
do I = 1, N
  do J = 1, min(I, N-1)
    do K = J, min(I, N-1)
      loop body
enddo
```

(b) Transformed loop nest

```
do J = 1, min(N, N-1)
  do K = J, min(N, N-1)
    do I = max(1, J, K), N
      loop body
enddo
```

Figure 4.17: (a) Original loop nest. (b) Loop nest after applying a loop permutation transformation. The exact loop bounds have been computed using the Fourier-Motzkin algorithm.

To eliminate these non-trivial redundant bounds, several researchers [5][7][16] propose the use of the FM algorithm to check if a simple bound is redundant with respect to the potential values of outer loops. The idea consists in creating a system of inequalities with all the simple bounds of outer loops, replacing the simple bound to be checked by its negation. If the new system is inconsistent, the simple bound is redundant and can be eliminated (to check if a system is inconsistent, the FM algorithm is used). Therefore, to eliminate all non-trivial redundant simple bounds of a loop, they apply the Fourier-Motzkin algorithm as many times as the number of simple bounds present in the loop. Thus, the complexity of eliminating redundant bounds is the complexity of executing the FM algorithm as many times as bounds have to be checked. As it can be observed, this technique, named *Exact Simplification* in [16], is very time-consuming.

Let's now see both why our implementation of Simultaneous Multilevel Tiling generates fewer non-trivial redundant bounds than a conventional implementation and why our implementation allows reducing the cost of eliminating the remaining redundant bounds in the innermost loops when the Exact Simplification technique is used.

Generating Fewer Non-trivial Redundant Bounds

Simultaneous Multilevel Tiling generates less redundant bounds than conventional techniques due to the way in which the bounds of the EL-loops are computed. Non-trivial redundant simple bounds can be generated by conventional implementations from the second loop permutation transformation performed onwards. They are generated when a loop `I` is moved inside another loop `J` that has been previously strip-mined and `J` has simple bounds that are affine functions of `I`. Moreover, a redundant bound generated due to a loop permutation is propagated to other loops in later loop permutations.

By contrast, our implementation of SMT does not generate these particular non-trivial redundant simple bounds, because it always computes the bounds of an EL-loop before applying strip-clustering to the C-loops associated to not-yet-processed EL-loops.

<table>
<tr><th colspan="3">Types of Redundant Simple Bounds
(how is their redundancy deduced)</th></tr>
<tr><th>Trivial</th><th colspan="2">Non-Trivial</th></tr>
<tr><td rowspan="3">looking at the loop's own simple bounds</td><td colspan="2">looking at the simple bounds of outer loops</td></tr>
<tr><td></td><td>Special</td></tr>
<tr><td></td><td>looking at the inequalities imposed by the strip-mining or strip-clustering transformation</td></tr>
</table>

Table 4.1: Types of redundant simple bounds according to the way their redundancy is deduced.

Let's now examine these assertions more formally. We start by giving a definition to characterize a special type of non-trivial redundant bound that appears in the multilevel tiling context and we present a lemma that shows how this special type of redundant bounds are generated. Then, we give a corollary showing how other non-trivial (and non-special) redundant bounds can additionally be created. Finally, we enounce a theorem that shows why the SMT implementation generates fewer redundant bounds than a conventional implementation of multilevel tiling.

Definition 4

Let II be a TI-loop associated to the EL-loop I and let B_{II} be the strip size of II. By definition of strip-mining (or strip-clustering), loop I always iterates inside the tiles determined by II. Therefore the inequalities $II \leq I$ and $I \leq II+B_{II}-1$ always hold. We refer as *special redundant simple bounds* to non-trivial redundant simple bounds whose redundancy can be deduced by looking at the inequalities imposed by previously applied strip-mining (or strip-clustering) transformations (Table 4.1 summarizes the different types of redundant bounds according to the way their redundancy is deduced).

Lemma 4

Let I be a loop that has been strip-mined and has simple bounds that are affine functions of J. Then, moving loop J inside loop I produces *special redundant simple bounds* in loop J.

Proof

Let's consider the following loop nest:

$$\begin{array}{l} \text{do } J = L, U \\ \quad \text{do } I = \alpha_1 \cdot J + \theta_1 , \alpha_2 \cdot J + \theta_2 \\ \quad\quad \textit{loop body} \\ \text{enddo} \end{array} \qquad (1)$$

where $\alpha_1, \alpha_2, \theta_1, \theta_2$, L and U are integer constants or program parameters (variables unchanged within the loops). Let assume that $\alpha_1, \alpha_2 > 0$ and $\alpha_1 \leq \alpha_2$. For other values of α_1 and α_2 the demonstration would be done in a similar way. We also assume that loop nest (1) has exact bounds, therefore the inequality $L \geq \left\lceil \frac{\theta_1 - \theta_2}{\alpha_2 - \alpha_1} \right\rceil$ holds.

To demonstrate Lemma 4 we will first apply strip-mining to loop I, second we will permute loops J and II (the TI-loop generated in the strip-mining transformation) and third, we will perform another loop permutation to move loop J to the innermost position[13]. For simplicity and without loss of generality, we use a null offset in the strip-mining transformation.

We perform this sequence of transformations because, in a later theorem (Theorem 5), we will use Lemma 4 to demonstrate that a conventional tiling implementation generates redundant bounds. In a conventional implementation, an EL-loop is never moved inside a TI-loop (recall that, by construction, the EL-loops are always the innermost loops in the nest) but EL-loops can be moved inside other EL-loops that have been previously strip-mined. In this demonstration we will see that moving an EL-loop inside another EL-loop that has been previously strip-mined (the second loop permutation) produces *special redundant simple bounds*.

By definition, every time strip-mining is applied to a loop, it is decomposed into two loops (the TI-loop and the EL-loop) and new simple bounds appear (recall Fig. 4.5, Section 4.3.2). After strip-mining loop I of (1) the following code is obtained:

$$\begin{array}{l} \text{do } J = L, U \\ \quad \text{do } II = \lfloor (\alpha_1 \cdot J + \theta_1) / B_{II} \rfloor * B_{II}, \alpha_2 \cdot J + \theta_2 , B_{II} \\ \quad\quad \text{do } I = \max (II, \alpha_1 \cdot J + \theta_1), \min(II + B_{II} - 1, \alpha_2 \cdot J + \theta_2) \\ \quad\quad\quad \textit{loop body} \\ \text{enddo} \end{array} \qquad (2)$$

13.Note that we could perform a loop permutation only once to move loop J to the innermost position, instead of performing a loop permutation twice as mentioned above. However, we will perform a loop permutation twice to show that redundant bounds are generated when loop J is moved inside loop I (the second loop permutation) but not when J is moved inside loop II.

Now, loop I in (2) has the same simple bounds as in (1) plus two new simple bounds: the lower simple bound (II) and the upper simple bound (II+B_{II}-1). Thus, the inequalities $II \le I$ and $I \le II+B_{II}-1$ hold.

Let's now perform a loop permutation to move loop J inside loop II. To perform the loop permutation we apply Theorem 1 (Section 4.3.3), because the iteration space defined by loops J and II in (2) is non-convex. Recall that Theorem 1 uses the FM algorithm to compute the exact bounds in the transformed code and we note that *trivial redundant bounds* will be directly eliminated. After the loop permutation, the following code is obtained (note that this loop permutation does not produce any redundant bound):

$$\begin{array}{l}
\text{do II} = \lfloor (\alpha_1 \cdot L + \theta_1)/B_{II} \rfloor * B_{II},\ \alpha_2 \cdot U + \theta_2,\ B_{II} \\
\quad \text{do J} = \max(L, \left\lceil \frac{II - \theta_2}{\alpha_2} \right\rceil), \min(U, \left\lfloor \frac{II + B_{II} - 1 - \theta_1}{\alpha_1} \right\rfloor) \\
\quad\quad \text{do I} = \max(II, \alpha_1 \cdot J + \theta_1), \min(II+B_{II}-1, \alpha_2 \cdot J + \theta_2) \\
\quad\quad\quad \textit{loop body} \\
\text{enddo}
\end{array} \qquad (3)$$

Let's now perform a loop permutation to move loop J to the innermost position. Now, to perform the loop permutation, we can directly apply the theory of unimodular transformations [121] because the iteration space defined by loops J and I in (3) is convex. After the loop permutation, the following code is obtained:

$$\begin{array}{l}
\text{do II} = \lfloor (\alpha_1 \cdot L + \theta_1)/B_{II} \rfloor * B_{II},\ \alpha_2 \cdot U + \theta_2,\ B_{II} \\
\quad \text{do I} = \max(II, \alpha_1 \cdot L + \theta_1, \left\lceil \frac{\alpha_1 \cdot (II - \theta_2)}{\alpha_2} + \theta_1 \right\rceil), \min(II+B_{II}-1, \alpha_2 \cdot U + \theta_2, \left\lfloor \frac{\alpha_2 \cdot (II + B_{II} - 1 - \theta_1)}{\alpha_1} + \theta_2 \right\rfloor) \\
\quad\quad \text{do J} = \max(L, \left\lceil \frac{I - \theta_2}{\alpha_2} \right\rceil, \left\lceil \frac{II - \theta_2}{\alpha_2} \right\rceil), \min(U, \left\lfloor \frac{I - \theta_1}{\alpha_1} \right\rfloor, \left\lfloor \frac{II + B_{II} - 1 - \theta_1}{\alpha_1} \right\rfloor) \\
\quad\quad\quad \textit{loop body} \\
\text{enddo}
\end{array} \qquad (4)$$

Now, loop J has two simple bounds (one upper and one lower simple bound) that depend on I, namely $J \le \left\lfloor \frac{I - \theta_1}{\alpha_1} \right\rfloor$ and $J \ge \left\lceil \frac{I - \theta_2}{\alpha_2} \right\rceil$. These bounds have been obtained in the following manner: before the loop permutation (code (3)), the relations between I and J were described by the simple bounds $\alpha_1 \cdot J + \theta_1$ and $\alpha_2 \cdot J + \theta_2$ of I. Then, the FM algorithm solved these relations $I \ge \alpha_1 \cdot J + \theta_1$ and $I \le \alpha_2 \cdot J + \theta_2$ for J, obtaining the bounds $\left\lfloor \frac{I - \theta_1}{\alpha_1} \right\rfloor$ and $\left\lceil \frac{I - \theta_2}{\alpha_2} \right\rceil$ of J (code (4)).

Loop J also has two simple bounds that depend on II, namely $J \le \left\lfloor \frac{II + B_{II} - 1 - \theta_1}{\alpha_1} \right\rfloor$ and $J \ge \left\lceil \frac{II - \theta_2}{\alpha_2} \right\rceil$. These bounds were obtained in the first loop permutation (code (2)) by solving J from $II \ge \alpha_1 \cdot J + \theta_1 - B_{II} + 1$ and $II \le \alpha_2 \cdot J + \theta_2$.

Since I and II were obtained by strip-mining loop I in the original code, the inequalities $II \le I$ and $I \le II + B_{II} - 1$ hold and therefore:

$$\max\left(\left\lceil \frac{II - \theta_2}{\alpha_2} \right\rceil, \left\lceil \frac{I - \theta_2}{\alpha_2} \right\rceil\right) = \left\lceil \frac{I - \theta_2}{\alpha_2} \right\rceil$$

$$\min\left(\left\lfloor \frac{II + B_{II} - 1 - \theta_1}{\alpha_1} \right\rfloor, \left\lfloor \frac{I - \theta_1}{\alpha_1} \right\rfloor\right) = \left\lfloor \frac{I - \theta_1}{\alpha_1} \right\rfloor$$

Thus, the simple bounds $\left\lfloor \frac{II + B_{II} - 1 - \theta_1}{\alpha_1} \right\rfloor$ and $\left\lceil \frac{II - \theta_2}{\alpha_2} \right\rceil$ of J in code (4) are redundant bounds. Moreover, they are *special redundant bounds* because (1) they are non-trivial (it is necessary to look at the simple bounds of loop I to deduce that they are redundant) and (2) their redundancy can be deduced by looking at the inequalities imposed by the strip-mining transformation.■

Corollary 3

Let I be a loop that has been strip-mined and has, at least, one lower simple bound and one upper simple bound that are affine functions of another loop J. Then, moving loop J inside loop I produces *non-trivial redundant simple bounds* in loop I.

Proof

In the second step of the Fourier-Motzkin algorithm, each of the lower simple bounds of the iteration variable being solved is compared with each of the upper simple bounds. These comparisons generate inequalities that might become new simple bounds of other loops.

In (4), the bounds $\left\lceil \frac{\alpha_1 \cdot (II - \theta_2)}{\alpha_2} + \theta_1 \right\rceil$ and $\left\lfloor \frac{\alpha_2 \cdot (II + B_{II} - 1 - \theta_1)}{\alpha_1} + \theta_2 \right\rfloor$ of I were generated by comparing each lower simple bound of J with each one of its upper simple bounds. More precisely, they were generated by comparing $\left\lceil \frac{II - \theta_2}{\alpha_2} \right\rceil \le \left\lfloor \frac{I - \theta_1}{\alpha_1} \right\rfloor$ and $\left\lceil \frac{I - \theta_2}{\alpha_2} \right\rceil \le \left\lfloor \frac{II + B_{II} - 1 - \theta_1}{\alpha_1} \right\rfloor$, respectively.

Taking into account that:

$$L \geq \left\lceil \frac{\theta_1 - \theta_2}{\alpha_2 - \alpha_1} \right\rceil \text{ and } II \geq \alpha_1 \cdot L + \theta_1 - B_{II} + 1$$

it can be trivially deduced that:

$$\max(II, \left\lceil \frac{\alpha_1 \cdot (II - \theta_2)}{\alpha_2} + \theta_1 \right\rceil) = II$$

$$\min(II+B_{II}-1, \left\lfloor \frac{\alpha_2 \cdot (II + B_{II} - 1 - \theta_1)}{\alpha_1} + \theta_2 \right\rfloor) = II+B_{II}-1$$

Therefore the simple bounds $\left\lceil \frac{\alpha_1 \cdot (II - \theta_2)}{\alpha_2} + \theta_1 \right\rceil$ and $\left\lfloor \frac{\alpha_2 \cdot (II + B_{II} - 1 - \theta_1)}{\alpha_1} + \theta_2 \right\rfloor$ of I in (4) are redundant. Moreover, they are *non-trivial redundant bounds* because it is necessary to look at the simple bounds of loop II to deduce that they are redundant. However, they are not *special redundant bounds* because their redundancy cannot be deduced by looking at the inequalities imposed by the strip-mining transformation.

Finally, these redundant bounds have been generated in the second step of the Fourier-Motzkin algorithm, by comparing $\left\lceil \frac{II - \theta_2}{\alpha_2} \right\rceil \leq \left\lfloor \frac{I - \theta_1}{\alpha_1} \right\rfloor$ and $\left\lceil \frac{I - \theta_2}{\alpha_2} \right\rceil \leq \left\lfloor \frac{II + B_{II} - 1 - \theta_1}{\alpha_1} \right\rfloor$. Thus, these redundant bounds can only be generated if there are at least one lower simple bound and one upper simple bound in the original code that are affine functions of loop J. ■

So far, we have seen different situations in which *special redundant simple bounds* and other *non-trivial redundant bounds* can be generated. Now, we will show why the SMT implementation generates fewer redundant bounds than a conventional multilevel tiling implementation. In particular, we will see that the SMT implementation does not generate *special redundant bounds* and other *non-trivial redundant bounds*, while a conventional implementation generates them.

Theorem 5

The SMT implementation generates fewer non-trivial redundant simple bounds in the tiled code than a conventional implementation of multilevel tiling.

Proof

Let the following loop nest be the original code:

$$\begin{array}{l}\texttt{do J = L, U}\\ \quad \texttt{do I = } \alpha_1 \cdot \texttt{J} + \theta_1 \texttt{, } \alpha_2 \cdot \texttt{J} + \theta_2 \\ \quad\quad \textit{loop body}\\ \texttt{enddo}\end{array} \qquad (5)$$

where $\alpha_1, \alpha_2, \theta_1, \theta_2$, L and U are integer constants or program parameters (variables unchanged within the loops). Let's assume that $\alpha_1, \alpha_2 > 0$ and $\alpha_1 \leq \alpha_2$. For other values of α_1 and α_2 the demonstration would be done in a similar way. We also assume that loop nest (5) has exact bounds, therefore the inequality $\texttt{L} \geq \left\lceil \frac{\theta_1 - \theta_2}{\alpha_2 - \alpha_1} \right\rceil$ holds.

To demonstrate Theorem 5 we will first apply loop tiling using a conventional implementation and we will see how non-trivial redundant bounds are generated from the second loop permutation transformation performed onwards. Then, we will apply loop tiling using the SMT implementation and we will see that these non-trivial redundant bounds are not generated.

Let assume that the desired loop order in the final tiled code is II-JJ-I-J. A conventional implementation applies strip-mining and loop permutation repeatedly until the desired code is obtained (see Section 4.2), computing the bounds of the loops in the final code from outermost to innermost. For simplicity and without loss of generality, we use null offsets in the strip-mining transformation.

Let's first apply strip-mining to loop I and perform a loop permutation of loops I and J using the FM algorithm[14]. In the loop permutation, we are moving loop I inside loop J and loop J has not yet been strip-mined. Thus, this loop permutation does not introduce any *special redundant bound*. After these transformations, the following loop nest is obtained (*trivial redundant bounds* have been directly eliminated):

$$\begin{array}{l}\texttt{do II=}\left\lfloor (\alpha_1 \cdot \texttt{L} + \theta_1)/\texttt{B}_{\texttt{II}} \right\rfloor \texttt{*B}_{\texttt{II}}\texttt{, } \alpha_2 \cdot \texttt{U} + \theta_2 \texttt{, B}_{\texttt{II}} \\ \quad \texttt{do J = max (L, } \left\lceil \frac{\texttt{II} - \theta_2}{\alpha_2} \right\rceil \texttt{), min(U, } \left\lfloor \frac{\texttt{II} + \texttt{B}_{\texttt{II}} - 1 - \theta_1}{\alpha_1} \right\rfloor \texttt{)} \\ \quad\quad \texttt{do I = max(II, } \alpha_1 \cdot \texttt{L} + \theta_1 \texttt{, } \alpha_1 \cdot \texttt{J} + \theta_1 \texttt{), min(II+B}_{\texttt{II}}\texttt{-1, } \alpha_2 \cdot \texttt{U} + \theta_2 \texttt{, } \alpha_2 \cdot \texttt{J} + \theta_2 \texttt{)} \\ \quad\quad\quad \textit{loop body} \\ \texttt{enddo}\end{array} \qquad (6)$$

14.Note that, before strip-mining loop I, it is necessary to perform a loop permutation in the original code because the original loop order is not such that the outermost loop is the loop to be strip-mined first.

Let's now apply strip-mining to loop J in (6) and perform again a loop permutation of loops J and I using the FM algorithm. After performing these transformations the following code is obtained:

$$
\begin{aligned}
&\text{do } II = \lfloor (\alpha_1 \cdot L + \theta_1)/B_{II} \rfloor^{*} B_{II},\ \alpha_2 \cdot U + \theta_2,\ B_{II} \\
&\quad \text{do } JJ = \lfloor \max\left(L, \left\lceil \frac{II-\theta_2}{\alpha_2} \right\rceil\right)/B_{JJ} \rfloor^{*} B_{JJ},\ \min\left(U, \left\lfloor \frac{II+B_{II}-1-\theta_1}{\alpha_1} \right\rfloor\right),\ B_{JJ} \qquad (7) \\
&\qquad \text{do } I = \max\left(II,\ \alpha_1 \cdot L + \theta_1,\ \alpha_1 \cdot JJ + \theta_1,\ \left\lceil \frac{(II-\theta_2)\cdot\alpha_1}{\alpha_2} + \theta_1 \right\rceil\right), \\
&\qquad\qquad \min\left(II+B_{II}-1,\ \alpha_2 \cdot U + \theta_2,\ \alpha_2 \cdot (JJ + B_{JJ} - 1) + \theta_2,\ \left\lfloor \frac{\alpha_2 \cdot (II + B_{II} - 1 - \theta_1)}{\alpha_1} + \theta_2 \right\rfloor\right) \\
&\qquad\quad \text{do } J = \max\left(JJ, L, \left\lceil \frac{II-\theta_2}{\alpha_2} \right\rceil, \left\lceil \frac{I-\theta_2}{\alpha_2} \right\rceil\right), \min\left(JJ+B_{JJ}-1, U, \left\lfloor \frac{II+B_{II}-1-\theta_1}{\alpha_1} \right\rfloor, \left\lfloor \frac{I-\theta_1}{\alpha_1} \right\rfloor\right) \\
&\qquad\qquad \textit{loop body} \\
&\text{enddo}
\end{aligned}
$$

In this loop permutation, loop J has been moved inside loop I that has already been strip-mined. From Lemma 4 and Corollary 3 this loop permutation generates *special redundant bounds* in loop J and other *non-trivial redundant bounds* in loop I. More precisely, the following bounds in (7) are redundant:

- the simple bounds $\left\lceil \frac{II-\theta_2}{\alpha_2} \right\rceil$ and $\left\lfloor \frac{II+B_{II}-1-\theta_1}{\alpha_1} \right\rfloor$ of J are *special redundant bounds* and
- the simple bounds $\left\lfloor \frac{II+B_{II}-1-\theta_1}{\alpha_1} \right\rfloor$ and $\left\lfloor \frac{\alpha_2 \cdot (II+B_{II}-1-\theta_1)}{\alpha_1} + \theta_2 \right\rfloor$ of I are *non-trivial redundant bounds.*

Finally, it can be trivially deduced that if another level of tiling is performed in (7) the new TI-loops will inherit these redundant bounds.

Let's now apply loop tiling to the original code (5) using the SMT implementation. The SMT implementation computes the bounds in the tiled code from the innermost to the outermost loops, applying the strip-clustering transformation as late as possible, that is, just before the bounds of a loop have to be computed. Again, for simplicity and without loss of generality, we use null offsets in the strip-clustering transformation.

The SMT implementation starts computing the bounds of J: first, strip-clustering is applied to C-loop J of the original code (5), obtaining the EL-loop J and the new C-loop JJ; and second, one iteration of the FM algorithm is performed to compute the bounds of J in the tiled code. After these transformations, the following loop bounds are obtained (recall that the SMT implementation works with the bounds of the MCS of the NCBIS to obtain the bounds of the MCS of the BTIS and we use braces on the loop bounds to indicate it):

strip-clustering loop J

do JJ = { L-B_{JJ}+1, U }

do J = { max(JJ, L), min(JJ+B_{JJ}-1, U) }

do I = { $\alpha_1 \cdot J + \theta_1$, $\alpha_2 \cdot J + \theta_2$ }

computing bounds of J

do JJ = { L-B_{JJ}+1, U }

do I = { max($\alpha_1 \cdot L + \theta_1$, $\alpha_1 \cdot JJ + \theta_1$), min($\alpha_2 \cdot U + \theta_2$, $\alpha_2 \cdot (JJ + B_{JJ} - 1) + \theta_2$) }

do J = { max (JJ, L, $\left\lceil \frac{I - \theta_2}{\alpha_2} \right\rceil$), min(JJ+B_{JJ}-1, U, $\left\lfloor \frac{I - \theta_1}{\alpha_1} \right\rfloor$) }

Note that the bounds of J are computed before strip-clustering loop I. Thus, loop J does not have *special redundant bounds* and loop I does not either have any of the *non-trivial redundant bounds* that appeared in (7).

To obtain the final tiled code, strip-clustering has to be applied to loop I. Then, the bounds of JJ must be computed by performing one iteration of the FM algorithm (only the first step) and finally, the bounds of the loops must be corrected to obtain the exact bounds of the BTIS. The bounds of the loops after these transformations are the following:

strip-clustering loop I:

do JJ = { L-B_{JJ}+1, U }

do II = { max($\alpha_1 \cdot L + \theta_1$, $\alpha_1 \cdot JJ + \theta_1$)-B_{II}+1, min($\alpha_2 \cdot U + \theta_2$, $\alpha_2 \cdot (JJ + B_{JJ} - 1) + \theta_2$) }

do I = { max(II, $\alpha_1 \cdot L + \theta_1$, $\alpha_1 \cdot JJ + \theta_1$), min(II+B_{II}-1, $\alpha_2 \cdot U + \theta_2$, $\alpha_2 \cdot (JJ + B_{JJ} - 1) + \theta_2$) }

do J = { max (JJ, L, $\left\lceil \frac{I - \theta_2}{\alpha_2} \right\rceil$), min(JJ+B_{JJ}-1, U, $\left\lfloor \frac{I - \theta_1}{\alpha_1} \right\rfloor$) }

computing bounds of JJ:

do II = { $\alpha_1 \cdot L + \theta_1$-$B_{II}$+1, $\alpha_2 \cdot U + \theta_2$ }

do JJ = { max (L, $\left\lceil \frac{II - \theta_2}{\alpha_2} \right\rceil$)-B_{JJ}+1, min(U, $\left\lfloor \frac{II + B_{II} - 1 - \theta_1}{\alpha_1} \right\rfloor$) }

do I = { max(II, $\alpha_1 \cdot L + \theta_1$, $\alpha_1 \cdot JJ + \theta_1$), min(II+B_{II}-1, $\alpha_2 \cdot U + \theta_2$, $\alpha_2 \cdot (JJ + B_{JJ} - 1) + \theta_2$) }

do J = { max (JJ, L, $\left\lceil \frac{I - \theta_2}{\alpha_2} \right\rceil$), min(JJ+B_{JJ}-1, U, $\left\lfloor \frac{I - \theta_1}{\alpha_1} \right\rfloor$) }

correcting bounds:

$$\begin{array}{l}
\text{do } II = \lfloor (\alpha_1 \cdot L + \theta_1)/B_{II} \rfloor * B_{II},\ \alpha_2 \cdot U + \theta_2,\ B_{II} \\
\quad \text{do } JJ = \lfloor \max\left(L, \left\lceil \frac{II - \theta_2}{\alpha_2} \right\rceil\right)/B_{JJ} \rfloor * B_{JJ},\ \min\left(U, \left\lfloor \frac{II + B_{II} - 1 - \theta_1}{\alpha_1} \right\rfloor\right),\ B_{JJ} \\
\quad\quad \text{do } I = \max(II,\ \alpha_1 \cdot L + \theta_1,\ \alpha_1 \cdot JJ + \theta_1),\ \min(II + B_{II} - 1,\ \alpha_2 \cdot U + \theta_2,\ \alpha_2 \cdot (JJ + B_{JJ} - 1) + \theta_2) \quad (8) \\
\quad\quad\quad \text{do } J = \max\left(JJ, L, \left\lceil \frac{I - \theta_2}{\alpha_2} \right\rceil\right),\ \min\left(JJ + B_{JJ} - 1, U, \left\lfloor \frac{I - \theta_1}{\alpha_1} \right\rfloor\right) \\
\quad\quad\quad\quad \textit{loop body} \\
\text{enddo}
\end{array}$$

Note that the final tiled code (8) does not have the redundant bounds that appeared in (7). This is because SMT always computes the bounds of an EL-loop before applying strip-clustering to the C-loops associated to not-yet-processed EL-loops. Thus, an EL-loop is always moved inside other EL-loops that have not yet been strip-clustered.

Finally, we note that if several levels of tiling are performed using SMT, strip-clustering will be applied to C-loops having non-unit steps and, therefore, broadcasting phases will be performed. The broadcasting phase of strip-clustering adds new bounds to loops whose bounds have already been computed. However, these new bounds are never redundant bounds.■

Summarizing, moving outer loops inside the EL-loop of a previous strip-mined loop produces some particular non-trivial redundant simple bounds. In a conventional implementation, this situation of moving a loop inside the EL-loop of a previously strip-mined loop happens continuously. Moreover, the redundant bounds generated in a certain loop permutation are propagated to other loops in later loop permutations. Our implementation of Simultaneous Multilevel Tiling does not generate these particular non-trivial redundant bounds because it computes the bounds of the EL-loops before applying strip-clustering to the C-loops associated to not-yet-processed EL-loops.

Reducing the Cost of Eliminating the Remaining Redundant Bounds

As already mentioned, the elimination of redundant bounds in the EL-loops of the tiled code is important to generate efficient code and to reduce compile-time when the register level is being exploited. To this end, each of the simple bounds of the EL-loops has to be checked for redundancy using the *Exact Simplification* [16] technique. This technique checks if a simple bound is redundant with respect to the possible values of outer loops using the FM algorithm. Thus, the complexity of checking one simple bound for redundancy is the complexity of executing the FM algorithm that, as shown in previous sections, depends doubly exponentially on the number of loops involved. Then, the complexity of eliminating all redundant bounds in the EL-loops is the complexity of executing the FM algorithm as many times as bounds must be checked. From now on, we will refer to the task of eliminating all redundant bounds in the EL-loops as the *Exact Simplification Phase.*

Simultaneous Multilevel Tiling reduces the cost of eliminating the redundant bounds in the EL-loops (with respect to conventional implementations) for two reasons. On one hand, as previously mentioned, it generates less redundant bounds and, therefore, less bounds must be checked for redundancy.

On the other hand, the processing order of the loops in SMT (from innermost to outermost) allows performing the Exact Simplification Phase just after the bounds of the innermost EL-loops have been computed. Thus, the number of loops involved in each execution of the FM algorithm is reduced compared to performing the Exact Simplification Phase at the end of the multilevel tiling process.

Conventional techniques, however, compute the loop bounds in the final code from the outermost to the innermost loop[15]. Therefore, redundant bounds in the innermost loops cannot be eliminated until the multilevel tiling process has been finished. However, in conventional techniques, there are two alternatives to eliminate the redundant bounds of the final EL-loops: we can perform the Exact Simplification Phase at the end of the process (in this case all loops in the final code are involved in the executions of the FM algorithm) or we can perform the Exact Simplification Phase every time that a loop permutation is done (in this case the TI-loop iteration variables can be considered as constants and only the current EL-loops are involved in the execution of FM). Although this second alternative performs several times the Exact Simplification Phase, it is faster than the first one because, as already mentioned, the complexity of the Fourier-Motzkin algorithm depends doubly exponentially on the number of loops involved. Moreover, this second alternative avoids redundant bounds to be propagated in later loop permutation transformations.

Figure 4.18 illustrates when the Exact Simplification Phase is performed in both SMT and conventional implementations. For the conventional implementation we show the second alternative just given. Assume that the order of the loops in the original code is `K-J-I` (from outermost to innermost) and the desired loop order in the final tiled code is `JJJ-III-KK-JJ-II-K-J-I`. The white boxes indicate, for each iteration of the multilevel tiling process, the loop or loops whose bounds have been computed, and the grey ellipses indicate that the Exact Simplification Phase is performed at this point. The loops inside the ellipses are the loops involved in the executions of the FM algorithm and recall that only the bounds of the EL-loops are checked for redundancy. In a conventional implementation, the Exact Simplification Phase is performed as many times as loop permutations and only the EL-loops are involved in the executions of FM. In a SMT implementation the Exact Simplification Phase is performed only once but with double number of loops involved in the executions of FM.

15.Recall from Section 4.2 that conventional techniques implement multilevel tiling by applying tiling level by level, going from the outermost to the innermost level.

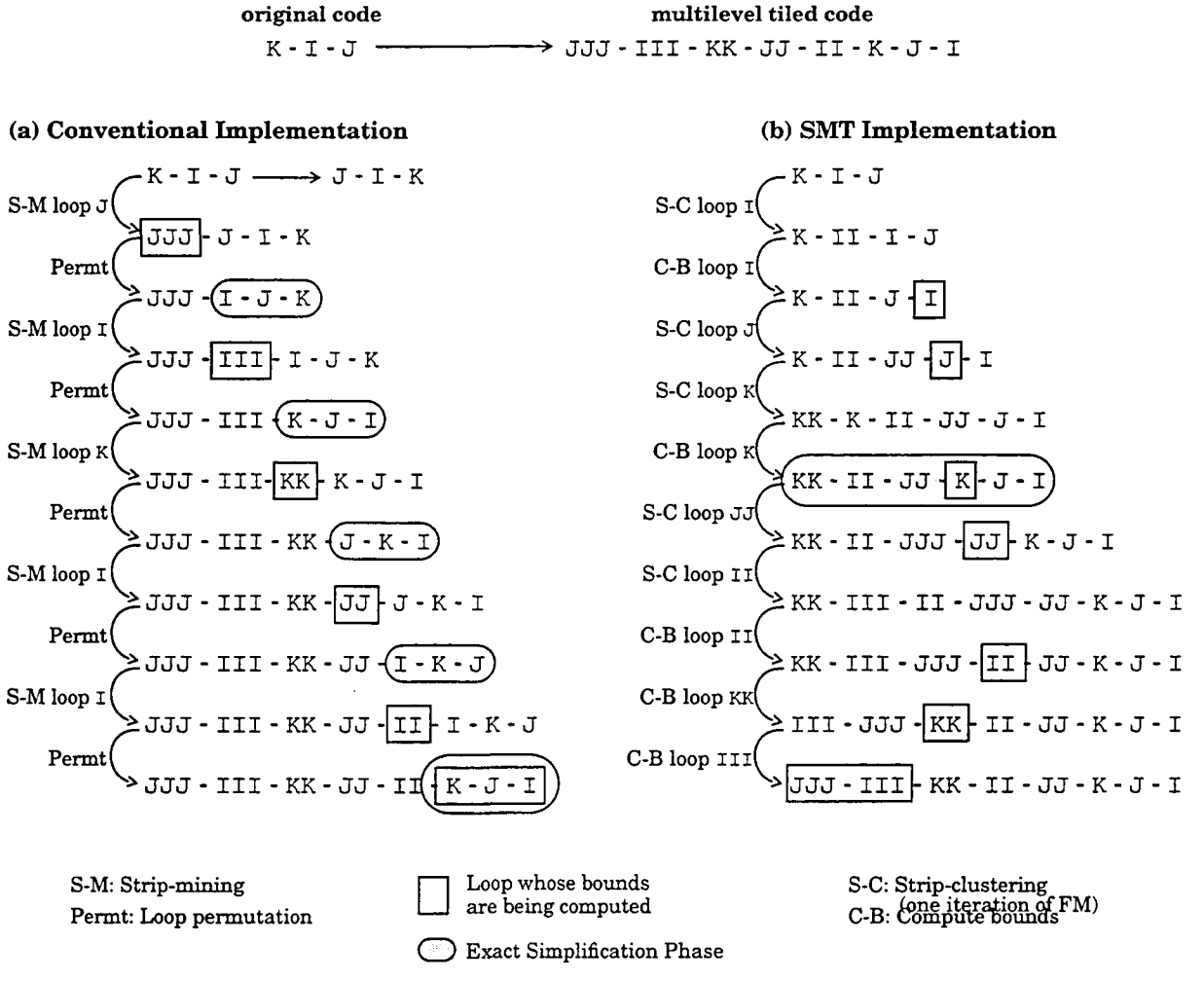

Figure 4.18: Exact Simplification Phase(s) in (a) a conventional implementation and (b) SMT.

Although the number of loops involved in the executions of FM is larger in SMT, the complexity of the overall process of eliminating the redundant bounds is smaller in SMT than in a conventional implementation. As shown in Section 4.2 the number of bounds in the EL-loops can increase doubly exponentially every time a loop permutation is performed in a conventional implementation. Therefore, the number of bounds to be checked for redundancy is much larger in a conventional implementation than in SMT. In the next section we will present some experimental results showing that SMT is also better than conventional implementations in that it allows removing the remaining redundant bounds in the innermost EL-loops at a much lower cost.

4.7 COMPARING SMT vs. CONVENTIONAL TECHNIQUES

In this section we compare SMT against conventional multilevel tiling techniques in terms of complexity, redundant bounds generated and cost of eliminating the remaining redundant bounds. For that purpose, we have implemented both SMT and conventional multilevel tiling. We note that both techniques were implemented such that they do not generate trivial redundant bounds. Therefore, all redundant bounds generated by both techniques are *non-trivial*. We have also implemented the Exact Simplification technique to measure the time required to eliminate the remaining redundant bounds in the innermost loops. As benchmark programs, we have used 12 linear algebra programs such as triangular matrix product, LU, Cholesky factorization, QR decomposition, SOR, Blas3 routines, etc. These loop nests are 3-deep and have 6 simple bounds with only one of them being affine function of one surrounding loop iteration variable. The measures were taken on a workstation with a SuperSPARC at 155 Mhz.

4.7.1 Complexity

Let's first see the significance of having a complexity that depends doubly exponentially on the number of loops in the original loop nest rather than in the number of loops in the tiled code. The worst-case complexity of both SMT and conventional implementations can be represented by a unique function, namely:

$$f(X, Y) = Y \cdot \left(\frac{q}{2}\right)^{2^X}$$

where q is the number of simple bounds in the original code. Using this formula, the complexity of SMT is given by $f(n\text{-}1, m\text{-}n)$ and the complexity of a conventional implementation is given by $f((m\text{-}n)\cdot(n\text{-}1), 1)$, where n and m are the number of loops in the original and tiled code, respectively.

Suppose that we have a 3-deep loop nest with 6 simple bounds and we perform 3 levels of tiling, obtaining an 8-deep tiled loop nest (that is, n=3, q=6 and m=8). For this particular example, the complexity value of SMT is $f(2,5)$ while that of a conventional implementation is $f(10,1)$. In Fig. 4.19 we have plotted the curves $f(X,1)$ and $f(X,5)$, with q fixed to 6 (note that they are overlapped) and we have also marked with points the complexity values of SMT and a conventional implementation for our particular example.

With this figure we want to outline that, even for small number of loops in the original code (n=3), there can be a very big difference in compile time. Although the depths of typical loop nests are usually small, after applying multilevel tiling the final loop nest has a larger number of loops. For example, tiling a 3-deep loop nest for 3 levels can yield an 8- or 9-deep loop nest. Therefore, even for small values of n, the SMT algorithm can perform much better than conventional techniques.

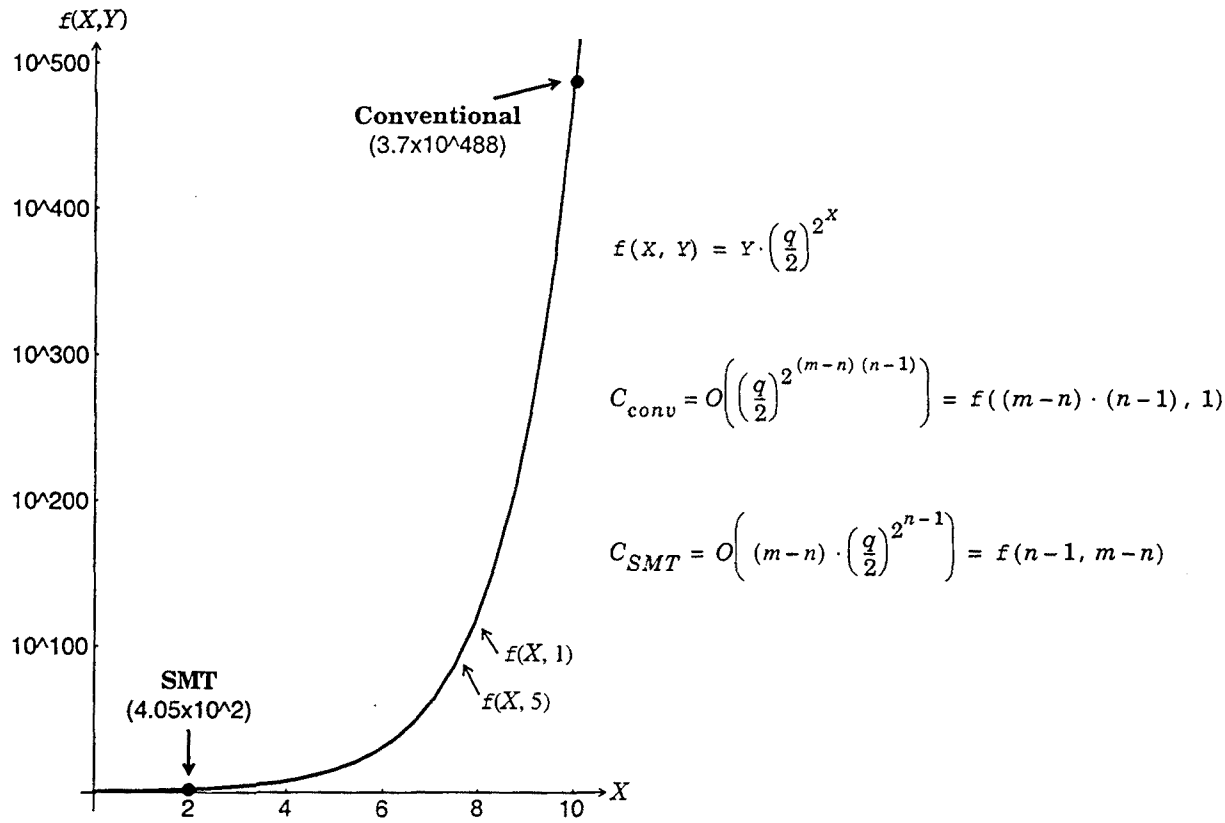

Figure 4.19: Complexity of SMT and conventional implementations. $f(2,5)$ and $f(10,1)$ represent C_{SMT} and C_{conv}, respectively, for the values n=3, m=8 and q=6.

Loop nests found in practice hardly ever incur in the worst-case complexity. However, we will next show that, for typical linear algebra programs, SMT significantly improves upon conventional implementations. We have measured the compile time required by each implementation to update the transformed code when tiling is applied, at different number of levels, for our benchmark programs. Figure 4.20 shows the average compilation time (in milliseconds) required by each technique, varying the levels of tiling from 1 to 4 (for each tile level we always partition two dimensions of the 3-dimensional iteration spaces). Also, Table 4.1 shows the average increment percentage of compilation time of a conventional implementation over SMT. It can be seen that, as expected, SMT behaves linearly with respect to the number of tile levels, while the conventional technique has an exponential behavior. This difference is very important since, if the number of memory levels in future architectures continues increasing, the ability to tile more levels will be critical to exploit the memory hierarchy efficiently.

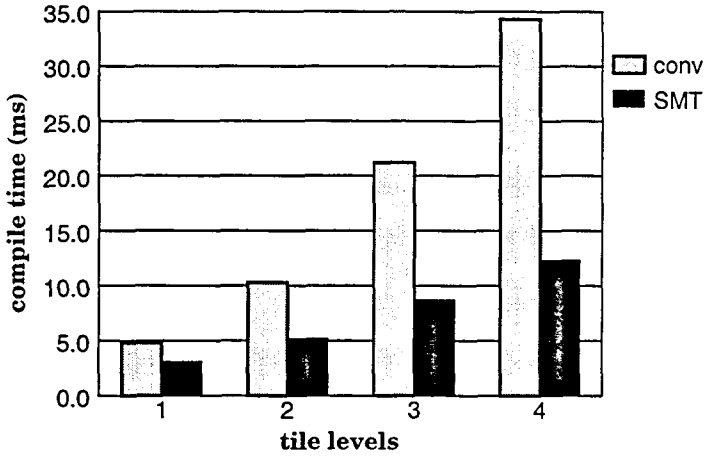

Figure 4.20: Average compile time (in milliseconds) of a conventional implementation and SMT for 12 linear algebra loop nests, varying the levels of tiling from 1 to 4.

Levels of Tiling	% increment (conv vs. SMT)
1	55%
2	77%
3	125%
4	181%

Table 4.1: Average increment percentage of compile time of a conventional implementation over SMT for different levels of tiling.

We also want to mention that for more complex loop nests, the difference in compile time is even more significant. As an example, after restructuring the rank 2k update SYR2K from BLAS as proposed by [84] to exploit both locality and parallelism, we obtain a 3-deep loop nest having 8 simple bounds with one of them being an affine function that depends on two loop index variables. The compile time of a conventional implementation is 10ms for one level of tiling and 32.67 seconds for four levels, while the compile time of SMT is 3.17ms for one level and 13.8ms for four levels. In this case, the compilation time increment of the conventional implementation over SMT varies from 215% for one level to over 200000% for four levels. At this point we want to outline that codes with bounds being affine functions that depend on two loop index variables, such as the SYR2K, are typically found in linear algebra programs that use banded matrices or can arise as a result of applying transformations such as loop skewing.

4.7.2 Redundant Bounds

Let's now present some data showing the number of redundant bounds generated by SMT and by conventional techniques and the cost of eliminating these redundant bounds in the innermost loops using the Exact Simplification technique in both implementations.

The FM algorithm used to compute the exact bounds of the loops can generate *non-trivial* redundant simple bounds [5]. As shown in Section 4.6, our SMT implementation does not generate some particular non-trivial redundant bounds, because it always computes the bounds of an EL-loop before applying strip-clustering to the C-loops associated to not-yet-processed EL-loops.

Let's first present some results showing how SMT generates less redundant bounds than conventional techniques. We have measured the total number of simple bounds generated by SMT and

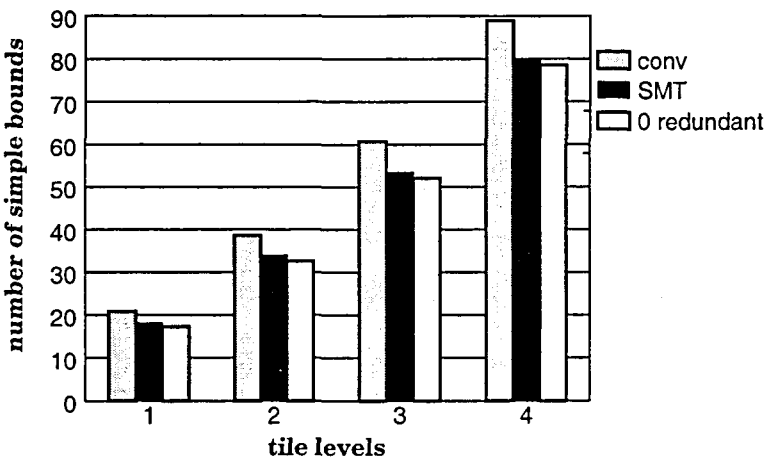

Figure 4.21: Average number of simple bounds generated by a conventional implementation and by SMT for 12 linear algebra loop nests, varying the levels of tiling from 1 to 4. We also show the average number of bounds if we eliminate all redundant bounds.

Tile levels	% redundant bounds	
	conv	SMT
1	17.1%	5.2%
2	15.5%	3.8%
3	14.2%	2.4%
4	11.7%	1.9%

Table 4.2: Average percentage of redundant bounds over the total number of bounds generated by a conventional implementation and by SMT, for different levels of tiling.

conventional implementations. Both implementations compute exact bounds, that means that, if we eliminate all redundant bounds, both implementations obtain the same final code. Figure 4.21 shows the average number of simple bounds in the final tiled code for the 12 programs, varying the levels of tiling from 1 to 4. We also show the average number of bounds if we eliminate all redundant bounds in the codes. It can be seen that SMT always generates less simple bounds than a conventional implementation and it almost does not generate redundant bounds. Table 4.2 shows the average percentage of redundant bounds over the total number of bounds generated by a conventional implementation and by SMT. For these loop nests, a conventional technique generates around 14% of redundant bounds while SMT only generates around 3.5%.

Again, we want to note that, for more complex loop nests such as SYR2K, the number of generated bounds in a conventional implementation can explode in an exponential manner. For the SYR2K example, the conventional implementation generates 61 simple bounds for one level of tiling and 2511 for four levels, while SMT only generates 24 for one level and 132 for four levels. Moreover, in this case, the percentage of redundant bounds over the total number of bounds generated by a conventional implementation varies from 60.6% for one level of tiling to 94.7% for four levels. However, using the SMT implementation, there is 0% of redundant bounds in the tiled code. Thus, for certain classes of loop nests, the number of generated (redundant) bounds in a conventional implementation can increase significantly.

As already mentioned, the presence of redundant bounds in the tiled code is negative if the register level is being exploited and, therefore, the elimination of redundant bounds, at least in the innermost loops, is important to generate efficient code and to reduce compile-time when the register level is being exploited. We have also measured the compile time required to eliminate these

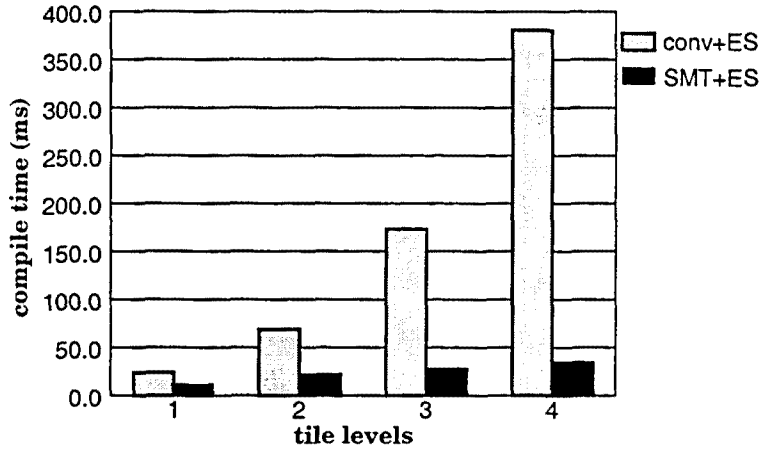

Figure 4.22: Average compile time (in milliseconds) required to compute the bounds and eliminate the redundant bounds of the innermost loops using the Exact Simplification technique in SMT and in a conventional implementation, for 12 linear algebra loop nests and varying the levels of tiling from 1 to 4.

Levels of Tiling	% increment (conv+ES vs. SMT+ES)
1	119%
2	216%
3	525%
4	1006%

Table 4.3: Average increment percentage of compile time of conv+ES over SMT+ES for different levels of tiling.

redundant bounds using the Exact Simplification technique. For that purpose we integrated the Exact Simplification technique in both SMT and conventional implementations and we measured the compile time required to both compute the bounds and eliminate the redundant bounds of the innermost loops. As mentioned in Section 4.6, the Exact Simplification Phase (ES) in SMT is performed just before the bounds of the TI-loops are going to be computed and, in conventional techniques, it is done every time that a loop permutation is performed.

Figure 4.22 shows the average compile time required by each new implementation (SMT+ES and conv+ES), varying the levels of tiling from 1 to 4 and, Table 4.3 shows the average increment percentage of compile time of conv+ES over SMT+ES. It can be seen that SMT+ES is significantly better than conventional techniques (SMT+ES is between 2.2 and 11 times faster than conv+ES) and, again, SMT+ES behaves linearly with respect to the number of tile levels, while the conventional technique has an exponential behavior. Thus, SMT is also better than conventional implementations in that it allows removing the remaining redundant bounds in the innermost loops (using the Exact Simplification technique) at a much lower cost.

4.8 RELATED WORK

There has been much research regarding loop tiling [23][27][28][98][110][122], that has been mostly focused on locality analysis (which loops have to be tiled and which is the loop order that yields best performance). Nonetheless, authors do not usually explain how the tiled loop nest is generated and the cost of computing the bounds of the tiled loop nest are not usually given. This cost analysis is very

important in order to determine if multilevel tiling is cost-effective enough to be implemented in a commercial compiler.

Several studies propose techniques to implement tiling for only one level. Multilevel tiling is then implemented by applying tiling level by level [28][121]. These techniques have very large complexities because they waste time computing loop bounds that will later be changed. For each level they compute the bounds of the TI and EL-loops. The bounds of the EL-loop, however, will be changed when tiling is applied in the next level. In this chapter, we have presented an implementation of multilevel tiling that deals with all levels simultaneously and that has a lower complexity.

M. Wolf and M. Lam in [123] present a method for determining the bounds of a loop nest after applying a unimodular transformation. The cost of the algorithm is linear in the number of loops and in the number of simple bounds. However, the resulting loop nest may contain redundant simple bounds and the loop bounds are not exact. When the register level is exploited using Index Set Splitting, redundant simple bounds produce a code explosion that can even prevent actual generation of the final code.

C. Ancourt and F. Irigoin in [7] propose a method to compute the exact loop bounds after tiling at one level but they do not evaluate precisely the complexity of their algorithm. If the method is extended directly to handle multilevel tiling, its complexity depends doubly exponentially on the number of loops in the tiled loop nest, while ours depends doubly exponentially on the number of loops in the original code. Their method works for any kind of tile shape while ours is restricted to rectangular tiles. However, most studies that focus on selecting an optimal tile shape typically end up using rectangular-shaped tiles, so we do not view this restriction as a shortcoming.

The presence of redundant bounds in the multilevel tiled code can be negative if the register level is being exploited. To eliminate redundant bounds, several researchers [5][7][16] propose the use of the Exact Simplification technique to eliminate all redundant bounds present in a loop nest. This technique is very time-consuming and could sometimes be not feasible to be implemented in a compiler. Our implementation of SMT generates less redundant bounds than traditional implementations and allows eliminating the remaining redundant bounds in the innermost loops (using the Exact Simplification technique) at a much lower cost than conventional implementations.

Finally, we note that A. Bik and H. Wijshoff in [16] and S. Amarasinghe in [5] present other low-cost methods to eliminate special non-trivial redundant simple bounds of a general loop nest. These methods can be used before the Exact Simplification Phase to reduce the number of bounds in the nest and therefore to reduce the cost of this phase.

4.9 SUMMARY

To improve the performance of a program the compiler usually applies multilevel tiling to maximize the effectiveness of the memory hierarchy and/or to reduce communication between processors. Moreover, the number of memory levels in today's and future computer architectures is continuously increasing and, thus, the ability to tile more memory levels is critical to exploit the memory hierarchy efficiently. Conventional techniques implement multilevel tiling by applying tiling level by level and their complexity depends doubly exponentially on the number of loops in the multilevel tiled code. This fact makes these techniques extremely costly when dealing with non-rectangular loop nests and when tiling for several levels.

In this chapter we have presented a new algorithm (SMT) to compute the exact loop bounds in multilevel tiling. We have first explained the theory to be able to perform multilevel tiling dealing with all levels simultaneously, and then we have proposed an efficient implementation of the technique, whose complexity depends doubly exponentially on the number of loops in the original loop nest; that is, it is proportional to the complexity of performing a loop permutation in the original code and thus, it is cost-effective enough to be implemented in a compiler. This is a very important achievement since, for example, it is very common to have 3-deep loop nests that have to be tiled in 3 levels (registers, cache and parallelism). This 3-deep loop nest turns into an 8-deep (or more) loop nest when multilevel tiling is applied. The complexity of our SMT algorithm is doubly exponential in 3 rather than in 8 (or more), which makes a big difference in compile time.

Further, the elimination of redundant bounds in the innermost loops is important to generate efficient code and to reduce compile-time when the register level is being exploited. We have also shown that the SMT algorithm avoids the generation of some particular non-trivial redundant simple bounds and allows eliminating the remaining redundant bounds efficiently.

Finally, we have compared our implementation of SMT against traditional techniques in terms of complexity, redundant bounds generated and cost of eliminating the remainder redundant bounds. We have shown that SMT is between 1.5 and 2.8 times faster than conventional implementations for simple non-rectangular loop nests, but it can be even 2300 times faster for more complex loop nests that are commonly found in linear algebra programs using banded matrices.

Moreover, experimental results also show that SMT generates less redundant bounds than conventional implementations and eliminating redundant bounds in a multilevel tiled code generated with SMT is between 2.2 and 11 times faster than in a code generated with conventional techniques. This is an important issue if the register level is being exploited. Overall, the efficiency of SMT makes it possible to integrate multilevel tiling including the register level in a production compiler without having to worry about compilation time.

5

MULTILEVEL TILING EVALUATION

Summary

This chapter is divided into two parts. In the first part, we present a detailed evaluation of loop tiling. We discuss and evaluate the effect of tiling for one memory level (either cache or registers) and then we study tiling for multiple memory levels. In the second part of this chapter, we compare our automatically-optimized codes against hand-optimized codes. The comparison shows that compiler technology can in most cases match the performance of hand-written vendor-supplied numerical libraries and that complex numerical codes are also able to achieve high performance on modern microprocessors.

5.1 INTRODUCTION

In Chapter 1, we showed that despite all the effort put into current compilers to achieve high performance in numerical codes, hand-optimized codes still outperform them. Moreover, the performance difference between hand-optimized codes and automatic-optimized codes is more noteworthy in complex numerical codes (loop nests defining non-rectangular iteration spaces). For non-rectangular loop nests, current compilers are not able to perform multilevel tiling and, thus, high performance is not achieved. The goal of this thesis was to show that multilevel tiling can also be applied to loops defining non-rectangular iteration spaces so that they achieve high performance on modern microprocessors. We bet for loop tiling to be the transformation that allows us to achieve high performance, because it is capable to achieve the three main optimizations required in modern microprocessors:

- improves a program's ILP, when it is applied at the register level,
- exploits data reuse in *several* dimensions of the iteration space and
- enhances data locality at *several* memory levels simultaneously (multilevel tiling).

In Chapter 3 and Chapter 4, we developed the compiler algorithms for applying multilevel tiling to non-rectangular loop nests. In this chapter, we will show that these algorithms can make possible that complex numerical codes achieve high performance on modern microprocessors.

This chapter is divided into two parts. In the first part, we will discuss the effects of tiling for different memory levels and we will also present quantitative data comparing the benefits of tiling only for the register level, tiling only for the cache level and tiling for both levels simultaneously. Since this thesis focuses on complex numerical codes, we use typical linear algebra algorithms having non-rectangular iteration spaces as benchmark programs. However, we note that the conclusions that we will draw out from this first part of the chapter also hold for loop nests defining rectangular iteration spaces.

In the second part of this chapter, we compare our automatically-optimized codes against the vendor hand-optimized codes and we show how compiler technology can make it possible for complex numerical codes to achieve high performance on modern microprocessors. More precisely, we will compare our automatically-optimized codes against the BLAS3 library[1] [38][39] on two different architectures, the ALPHA 21164 and the MIPS R10000.

1.The BLAS3 library provides a set of standard linear algebra operations which are highly optimized for each specific machine.

The remainder of this chapter is organized as follows: In Section 5.2 the effects of tiling for different memory levels are explained. Section 5.3 shows the evaluation process used in the following section. Section 5.4 presents quantitative data showing the benefits of tiling only for the register level, tiling only for the cache level and tiling for both levels simultaneously. In Section 5.5 we compare our automatically-optimized codes against the vendor hand-optimized codes. Finally, Section 5.6 presents the previous work related to memory hierarchy evaluation and in Section 5.7 we summarize this chapter.

5.2 EFFECTS OF LOOP TILING

Tiling is a loop transformation that has been mostly used to exploit data reuse at the different memory levels. Loop tiling exploits data reuse in several dimensions of the iteration space and enhances data locality at several memory levels simultaneously. Exploiting data reuse in more than one dimension of the iteration space, whenever possible, improves the performance of the memory hierarchy [121]. In particular, previous work on tiling stated that if in a n-dimensional iteration space all loops carry reuse, then n-1 loops should be tiled [97]. Not tiling one loop that carries data reuse and establishing a proper order of the inner loops yields bigger tile sizes and, therefore, more data locality than tiling all loops that carry reuse.

To exploit data reuse in several levels of the memory hierarchy simultaneously, Multilevel Tiling has to be performed. As shown in Chapter 4, Multilevel Tiling consists in recursively applying tiling to each level by dividing a tile of a higher level into subtiles [28][98][121]. Each level of tiling exploits one level of the memory hierarchy. When multilevel tiling is performed, the interaction between different levels must be considered. In this section, we present the effects of tiling for the register level, tiling for the cache level and tiling for both cache and register levels simultaneously.

5.2.1 Tiling for the Register Level

Tiling for the register level provides two different advantages: first, it exploits registers reuse and, second, it improves the intra-iteration ILP.

By tiling for the register level, values of recent memory access are held in registers so that data can be reused without accessing memory. Thus, data reuse at the register level essentially translates into a reduction of the number of load/store instructions. An important consequence of reducing the number of loads and stores instructions is an improvement in loop balance. Tiling for the register level improves the performance of memory-bound loops by converting them into balanced or compute-bound loops. Moreover, reducing the number of load/store instructions reduces the pressure on the fetch unit, reduces data memory traffic and, indirectly, might also decreases data cache misses. Futhermore, it can also lead to a reduction of the critical path length.

(a) Before tiling for the register level

```
do J = 1, N
  do K = 1, N
    do I = 1, N
      C(I,J) = C(I,J) + A(I,K) * B(K,J)
enddo
```

(b) After tiling for the register level

```
do JJ = 1, N, B_JJ
  do II = 1, N, B_II
    RR1= C(II,JJ)
    ...
    RRn= C(II+B_II-1,JJ+B_JJ-1)
    do K = 1, N
      RR1 = RR1 + A(II,K) * B(K,JJ)
      ...
      RRn = RRn + A(II+B_II-1,K) * B(K,JJ+B_JJ-1)
    enddo
    C(II,JJ)=RR1
    ...
    C(II+B_II-1,JJ+B_JJ-1)=RRn
enddo
```

Figure 5.1: Code of the square matrix product (a) before tiling for the register level and (b) after tiling for the register level, being K the non-tiled direction.

The second goal of tiling for the register level is to improve the ILP of the original loop nest by merging different iterations of the original loop body in a new, larger, loop body. In single loops, compilers can exhibit inter-iteration ILP by performing software pipelining and/or unrolling the loop. Nonetheless, they are limited by the recurrences of the loop body. In nested loops, however, there are more opportunities to expose ILP. In particular, tiling in more than one dimension for the register level always exposes ILP, regardless if the unrolled loops add new dependences or not in the new loop body. These two facts (exploiting register reuse and improving ILP) were illustrated in Chapter 3 (Section 3.3).

We have reviewed so far the benefits of tiling for the register level. However, tiling only for the register level has a drawback when dealing with large problem sizes. For large problem sizes, register tiling can heavily increase the overall TLB (Translation Lookaside Buffer) misses if spatial locality is not properly exploited, resulting in a performance degradation. More precisely, if there is at least one array reference that is traversed in row major order by the non-tiled loop, for these references spatial locality is not being exploited[2], and this fact makes TLB misses increase considerably.

To illustrate this increase in TLB misses we will use as an example the square matrix product code (Fig. 5.1a). We use a rectangular loop nest, instead of a non-rectangular loop nest, because the increase in TLB misses is caused by the data access patterns and not by the iteration space shape. This way, we simplify the explanation and the following figures.

2.We assume that programs are written in FORTRAN, that stores matrices in column major order.

(a) Before tiling for the register level **(b) After tiling for the register level**

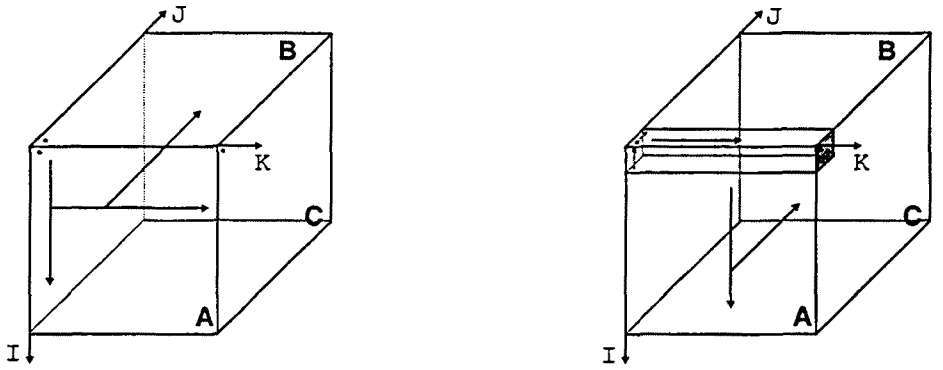

Figure 5.2: Data and Computation Diagram for the square matrix product, (a) before tiling for the register level, and (b) after tiling for the register level, being K the non-tiled direction.

To maximize register reuse, we will choose loop K as the non-tiled loop in the matrix product code. Note that one load and one store instruction (the references to C(I,J) (read) and to C(I,J) (write)) will be saved if loop K is the innermost loop in the nest. However, only one load instruction (the reference to A(I, K) or to B(K, J)) would be saved if loop J or loop I were the innermost loop in the nest. Thus, the loop that carries most temporal reuse is loop K. The tiled code of the matrix product is shown in Fig. 5.1b. B_{II} and B_{JJ} are the tile sizes in dimensions I and J, respectively, and, for simplicity, we assume N to be multiple of B_{II} and B_{JJ}.

To show the iteration space and the data access patterns before and after tiling for the register level (Fig. 5.2a and Fig. 5.2b, respectively), we use the Data and Computation Diagram (DCD) proposed in [98] as a very powerful visual tool to understand and to design multilevel tiled algorithms. In this diagram, the rectangular parallelepiped represents the iteration space, with the operations in the inside and the data in the faces or in planes parallel to these faces. The arrows indicate the order in which the data are accessed and the operations performed. To clarify data positions in this DCD, the elements A(1,1), B(1,1) and C(1,1) are represented in dark. The darkened portions in Fig. 5.2b show the matrix elements stored in registers at a given moment of the execution of the algorithm. From the DCD, it is apparent that matrix A can be reused in direction J (all iterations of loop J use the same element of A), matrix B can be reused in direction I and matrix C in direction K.

Before tiling for the register level and assuming the loop order J-K-I as shown in Fig. 5.2a, the three matrices A, B and C are accessed in column major order, thus exploiting spatial locality. The same element B(K,J) is used by all iterations of the innermost (I) loop; it can be register allocated and is fetched from memory only once. The same column of C accessed in the innermost loop (I) is reused in the next iteration of the middle (K) loop, and the same column of A is reused in the outermost (J) loop. Let's assume that the problem size (N) is so large that one page can only hold one matrix column. Unless the TLB has more than N+2 entries, there will be a TLB miss every time a column of A is reused. Thus, the number of TLB-misses in Fig. 5.2a is N^2+N+N.

After tiling for the register level as shown in Fig. 5.2b, the data access patterns have changed. In this case, the $B_{II}xB_{JJ}$ elements of C referenced in a register tile are reused by all iterations of the innermost (K) loop; they are register allocated and are fetched from memory only once. The same B_{JJ} columns of B accessed in the innermost loop (K) are reused in the next iteration of the middle (II) loop, and the same B_{II} elements of one column of A are reused in the next iteration of the outermost (JJ) loop. Moreover, references to the same column of A occur in each iteration of the middle (II) loop. Thus, unless the TLB has more than $N+2*B_{JJ}$ entries, there will be a TLB miss every time a column of A is referenced. The number of TLB misses in Fig. 5.2b is $(N^3/(B_{II}*B_{JJ}))+N+N$, which is about N times greater (assuming B_{II}, $B_{JJ}<<N$) than the non-tiled case. This high number of TLB misses can significantly degrade machine performance, since TLB misses have a large miss penalty.

Summarizing, tiling only for the register level exploits registers reuse and improves ILP. However, it can heavily increase the overall TLB misses if spatial locality is not properly exploited. We will see later in this chapter that this TLB problem can be solved by performing tiling for two memory levels, cache and registers.

5.2.2 Tiling for the Cache Level

For the cache level, tiling is effective for reducing the capacity cache miss rate and thus, potentially improving average memory access time. Moreover, it allows to state predictable latencies in memory instructions which can sometimes help in improving instruction scheduling. If a program is memory bound and a great portion of its total execution time is due to cache misses, then tiling for the cache level will be advantageous.

Let's illustrate how loop tiling enhances data locality at the cache level, using the square matrix product program (Fig. 5.1a and Fig. 5.2a). In this code, the same column of C is used repeatedly by iterations of the middle (K) loop. If N is large relative to the cache size, so that an entire column of an array does not fit into the cache, then elements of the column may not be in the cache between reuses. For the A matrix, reuses of array elements occur in the outermost (J) loop. Between reuses of elements in A, the whole array is brought into the cache, which means that references to A will not hit in the cache. Whether the data remains in the cache at the time of reuse depends on the size of the cache. Unless the cache is large enough to hold at least one NxN matrix, the data from A will have been displaced before reuse. If the cache cannot hold even one column of the NxN matrix, then C data in the cache will also not be reused. In this latter case (worst case), $2N^3+N^2$ words of data need to be read from main memory in N^3 iterations. This high ratio of memory fetches to numerical operations can significantly degrade machine performance, since memory fetches are of high latency.

Loop tiling alters the order in which individual iterations are executed so that iterations from loops of the outer dimensions are executed before completing all the iterations of the inner loop. Thus, the distance between successive references to the same memory location is shortened and the

```
do JJ = 1, N, B_JJ
  do II= 1, N, B_II
    do K = 1, N
      do J = JJ, min(N, JJ+B_JJ-1)
        do I = II, min(N, II+B_II-1)
          C(I,J) = C(I,J) + A(I,K) * B(K,J)
enddo
```

(a)

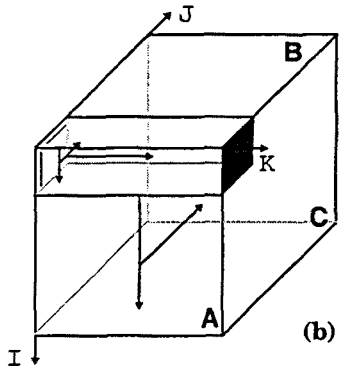

Figure 5.3: (a) Code of matrix product after tiling for the cache level. (b) Data and Computation Diagram for the matrix product after tiling for the cache level.

probability of finding the associated data in the cache is higher. Figure 5.3a shows the matrix product program after tiling for the cache level and Fig. 5.3b shows its corresponding DCD. The darkened portions show the matrix elements stored in cache at a given moment of the execution of the algorithm. The selection of the tile size B_{II} x B_{JJ} must attempt to maximize the cache utilization while eliminating (or reducing) self and cross interferences within the tile [31][42][80].

If B_{II} and B_{JJ} are chosen properly, the submatrix of C referenced inside a tile fits in the cache and can be reused over and over. Tiling for the cache level allows all three matrices to have excellent reuse; ignoring interferences in the cache, the total main-memory words accessed will be $(N^3/B_{II})+(N^3/B_{JJ})+N^2$, which is an improvement of about a factor of B_{II} (assuming $B_{II} \approx B_{JJ}$) over the non-tiled case.

Finally, we want to point out two issues related to the TLB when tiling is applied for the cache level. First, the TLB size must be considered when selecting the tile size at the cache level [90][91][98]. In particular, the number of TLB entries used by all data referenced inside a tile must not exceed the TLB size. In the example above, B_{JJ} columns of B, B_{JJ} columns of C and one column of A are referenced inside a tile. Therefore, $2*B_{JJ}+1$ must be smaller than the number of TLB entries (assuming one page can only hold one matrix column). If not, there will be a TLB miss in each iteration of the innermost EL-loops (loops J and I in Fig. 5.3), degrading machine performance significantly.

Second, as it happened for the register level, tiling only for the cache level can increase the overall TLB misses (with respect to the non-tiled code), if there is at least one array reference that is traversed in row major order by the non-tiled loop. In the example above, the number of TLB misses is $(N^3/(B_{II}*B_{JJ}))+N+N$, which is about $N/(B_{II}*B_{JJ})$ times greater than the non-tiled case (assuming $N>B_{II}*B_{JJ}$). However, the tile sizes for the cache level are much larger than the tile sizes for the register level and thus, this increase in overall TLB misses is not as significant as it was at the register level. Note that, if the factor $N/(B_{II}*B_{JJ})$ is less than 1, then the overall TLB misses of the tiled code will decrease with respect to the non-tiled code.

```
do JJJ = 1, N, B_JJJ
  do KKK= 1, N, B_KKK
    do II = 1, N, B_II
      do JJ = JJJ, JJJ+B_JJJ-1, B_JJ
        RR1= C(II,JJ)
        ...
        RRn= C(II+B_II-1,JJ+B_JJ-1)
        do K = KKK, KKK+B_KKK-1
          RR1 = RR1 + A(II,K) * B(K,JJ)
          ...
          RRn = RRn + A(II+B_II-1,K) * B(K,JJ+B_JJ-1)
        enddo
        C(II,JJ)=RR1
        ...
        C(II+B_II-1,JJ+B_JJ-1)=RRn
enddo
```

(a)

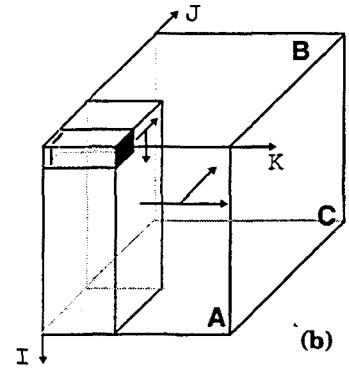

Figure 5.4: (a) Code of matrix product after tiling for cache and register levels. (b) Data and Computation Diagram for the matrix product after tiling for both levels.

5.2.3 Tiling for the Cache and Register Levels

When multilevel tiling is performed, the interaction between different levels must be considered [29][90]. Achieving the optimization of only one level of the hierarchy is simple, whereas the overall optimization for several levels is complex. In [97][98], Multilevel Orthogonal Block (MOB) forms are proposed to achieve a high degree of data reuse in all levels of the memory hierarchy. The basic rule in the construction of a MOB form is that the direction of blocks in adjacent levels should be different. The direction of a block is determined by the loop that is not tiled for this level. The orthogonality property of the MOB forms allows a "sequential" optimization to determine the order in which tiles are traversed and the size of the tiles level by level, beginning with the lowest level.

As an example, Fig. 5.4a shows the code of the matrix product program after tiling for cache and register levels using a MOB form and Fig. 5.4b shows its corresponding DCD. The grey portions show the matrix elements stored in cache at a given moment of the execution of the algorithm and the black portions show the matrix elements stored in registers. At the register level, loop K should be the non-tiled loop for maximizing register reuse. Then, to maximize the cache utilization, loop I has been selected as the non-tiled loop at this level. Note that MOB forms require loop K to be tiled at the cache level. B_{JJJ} and B_{KKK} are the tile sizes for the cache level in dimensions J and K, respectively, and, B_{II} and B_{JJ} are the tile sizes for the register level. For simplicity, we assume N to be multiple of the tile sizes and B_{JJJ} to be multiple of B_{JJ}.

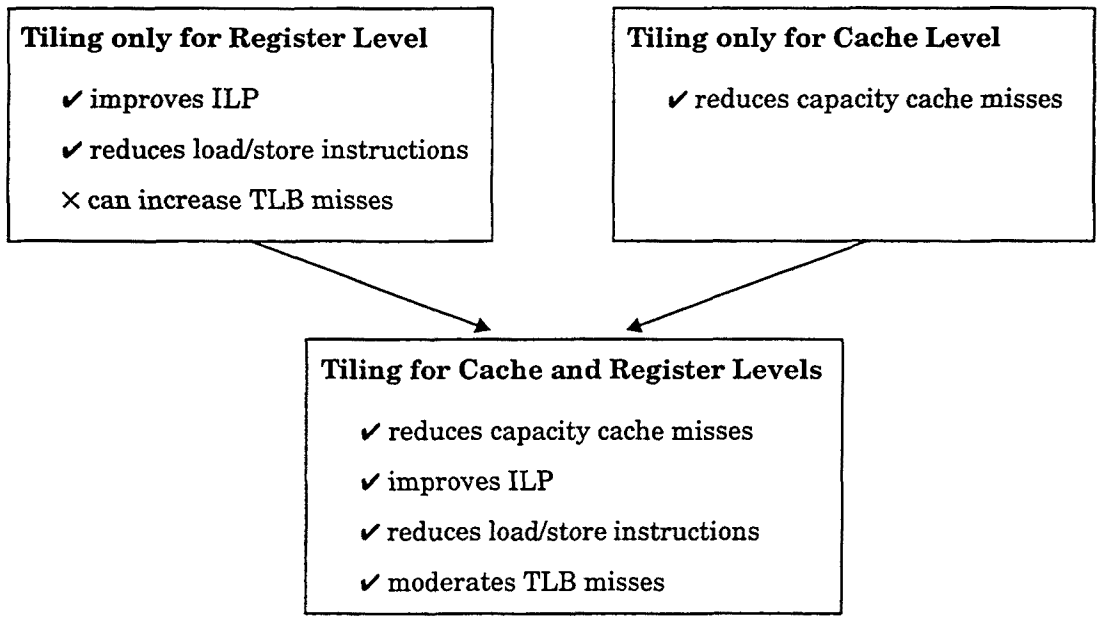

Figure 5.5: Effects of tiling for cache and register levels.

We have seen so far that tiling for the register level is effective to reduce memory instructions and to improve ILP, but it can significantly increase TLB misses. We have also seen that tiling for the cache level is effective for reducing cache misses. When tiling is performed for both levels simultaneously, we achieve both types of benefits. Moreover, we achieve another goal: TLB misses are moderated. This last goal is a consequence of using MOB forms.

Using these forms, all directions of the iteration space are bounded from the point of view of the register level, because the non-tiled direction at the register level must be tiled at the cache level. Thus, if the tile sizes at the cache level are properly chosen (considering the TLB size), the multilevel tiled code will not perform more TLB misses than the code tiled only for the cache level. Thus, TLB misses are moderated. In the example of Fig. 5.4 (assuming one page can only hold one matrix column and $2*B_{JJJ}+B_{KKK}$ is smaller than the number of TLB entries), the overall TLB misses is $(N^2/B_{JJJ})+N+N$, which is about B_{JJJ} times smaller than for the non-tiled case. Note that, in this case, there is no array reference that is traversed in row major order by the non-tiled loop selected for the cache level and thus, the overall TLB misses decrease (with respect to the non-tiled code).

To conclude this section, we have summarized in Fig. 5.5 the effects of tiling for cache and register levels. In the remainder of this chapter we present quantitative data showing all these effects.

Ref	Name	Description	perfect nested	affine bounds
[57]	MMtri	Triangular matrix product	Yes	2
[122]	LU	LU decomposition without pivoting	No	2
[27]	CHOL	Cholesky factorization	No	2
[121]	QR	Givens QR-decomposition	No	2
BLAS [39]	SSYR2K	symmetric rank 2k update	Yes	1
	SSYRK	symmetric rank k update	Yes	1
	SSYMM	symmetric matrix-matrix operation	No	1
	STRMM	product of triangular and square matrix	Yes	1
	STRSM	solve a matrix equation	No	1

Table 5.1: Description and characteristics of our benchmarks programs.

5.3 EVALUATION PROCESS

In this section, we will present our evaluation process. First, we describe the set of programs used as benchmarks; second, we give the main characteristics of the architectures where our measurements were taken; and third we describe how the different versions of the programs were generated.

5.3.1 Benchmarks Programs

Since this thesis focuses on non-rectangular loop nests, we have used as benchmark programs 9 linear algebra algorithms having non-rectangular, 3-dimensional iterations spaces. Table 5.1 contains the characteristics of each of them. Column labeled "Ref" indicates from where the algorithms were extracted. Five of them were extracted from the BLAS3 library [38][39], and the others were taken from several papers in the literature [27][57][121][122]. The third column gives a short description of the operation performed by each algorithm. The fourth column indicates whether the loops being transformed were perfectly nested or not. As pointed out in Chapter 2 (Section 2.3.3), for those programs having non-perfectly nested loops, we transformed them into a perfectly nested version using a code sinking transformation that was undone after loop tiling. Column labeled "affine bounds" indicates the total number of bound components in the original code that are affine functions of the surrounding loops iteration variables. The other bound components are integer or symbolic constants. Finally, we point out that, in program QR, we were forced to apply loop skewing, before tiling, to convert the loops into a fully permutable loop nest [121].

Architecture	MHz	issue rate (instr/cycle)	L/S per cycle	int/fp units	int/fp regs	L1-cache	L2-cache	TLB entries	Page size*
ALPHA 21164	266	4 (in-order)	2/1	2/2	32/32	8KB direct mapped	96KB 3-way	64	8KB
MIPS R10000	250	4 (out-of-order)	1/1	2/2	32/32	32KB 2-way	4MB 2-way	64	2x16KB

*. On the MIPS processor, one TLB entry maps two consecutive pages of 16KB.

Table 5.2: Characteristics of the architectures ALPHA AXP 21164 and MIPS R10000.

5.3.2 Target Architectures

All our measurements were taken on a uniprocessor system[3] with an ALPHA 21164 processor [15] and on a single R10000 processor [130] of a multiprocessor system[4] (SGI Origin 2000 [81]). The two different architectures are shortly described in Table 5.2. Note that the peak performance on the ALPHA processor is 532 Mflop/s and on the MIPS processor is 500 Mflop/s.

5.3.3 Code Generation

In Section 5.4 we will present quantitative data showing the effects of tiling only for the register level, tiling only for the cache level and tiling for both levels simultaneously. For that purpose, we evaluate four different versions of each program: one is the original version (UnOpt) with no restructuring transformation applied to it. The second one is the original code after tiling only for the register level (TRL). The third one corresponds to tiling only for the cache level (TCL) and the fourth one is the code after tiling for both cache and register levels (TCRL).

The versions TRL, TCL and TCRL of the codes were produced using our own developed tool that implements the method proposed in Chapter 3 to perform tiling for the register level and the technique proposed in Chapter 4 to compute exact loop bounds when multilevel tiling is applied. We note that this tool is able to manipulate symbolic expressions [5][6]. After generating the different versions for each program, we used the standard Fortran 77 compiler[5] to generate the final executables. Since we want to compare the effects of tiling for the different memory levels, the F77 compiler was used with the highest scalar optimization level that did not perform neither loop nest transformations nor software pipelining (-O4 on the ALPHA and -O2 on the MIPS). However, we note that, at these optimization levels, the F77 compiler unrolls the innermost loop when the loop body has a small number of operations in order to increase the instruction level parallelism.

3.The uniprocessor system runs version 4.0 of the DEC OSF/1 operating system.
4.The multiprocessor system runs version 6.5 of the IRIX64 operating system.
5.Version 4.1 of Digital Fortran on the ALPHA processor and version 7.2.1 of MIPSpro Compiler on the MIPS processor.

For the tiled versions (TRL, TCL and TCRL), we always tile, for each level, two dimensions of the 3-dimensional iteration spaces [98]. The non-tiled loop at the register level was selected using the heuristic proposed in Section 3.4.3 of Chapter 3. Tile sizes were chosen taking into account the available number of machine registers in order to reduce the register pressure and not overly constrain the job of the register allocator of the native compilers. When tiling for two levels (cache and registers) MOB forms were used. Thus, the non-tiled loop at the cache level is always one of the tiled loops at the register level. We select the one that provides better data locality. The tile sizes for the cache level were selected considering the available number of TLB entries and were such that less than 60% of the cache was used [31][80][97]. For the TCL version we use the same cache tiling parameters (tile sizes and non-tiled loop) as for the TCRL version. Finally, we want to stress that our goal is not finding the optimum tile sizes and shape for each level, but showing the effects of tiling for different memory levels. In particular, we want to show the additional benefit that can be achieved by tiling for the register level. Hence, we use the same tiles (size and shape) for different problem sizes and do not evaluate problem sizes that produce maximum self interference.

For the UnOpt version we always select the loop order that achieves, in average, the best performance. It is worth nothing that the best loop order for small problem sizes that fit in the cache level cannot be the best loop order for large problem sizes.

Table 5.3 summarizes for each program the tiled dimensions selected for the cache and register levels as well as the order of the loops used in our measures. For each program we show: the original loop order used in the UnOpt version (column 2), the tiled dimensions for the register level and their corresponding tile sizes (column 3), the loop order in the TRL version (column 4), the tiled dimensions for the cache level and their corresponding tile sizes (column 5), the loop order in the TCL version (column 6) and finally, the loop order in the TCRL version (column 7). In column 5 we give two different tile sizes, because our target architectures have different cache sizes. The tile sizes used on the ALPHA processor are labelled with "A" and the tile sizes used on the MIPS processor are labelled with "M". We use the iteration variables `IR`, `JR` and `KR` for the tile loops at the register level and `IC`, `JC` and `KC` for the tile loops at the cache level. The loops that are fully unrolled when tiling does include the register level are marked in bold. Finally, Table 5.4 shows how arrays are referenced in the main loop body of the original codes[6] and recall that Appendix C shows the original code of all our benchmark programs.

6.For the non-perfectly nested programs (LU, CHOL, QR, SSYMM and STRSM), we are not showing the additional statements outside the innermost loop.

Prog	Original loop order (UnOpt)	Register Level		Cache Level		Both Levels
		tile sizes	loop order (TRL)	tile sizes	loop order (TCL)	loop order (TCRL)
MMtri	K-J-I	(JxI)-(4x4)	JR-IR-K-**J**-I	(JxK)-(24x24)A (64x64)M	JC-KC-I-J-K	JC-KC-IR-JR-K-**J**-I
LU	K-J-I	(JxI)-(4x4)	JR-IR-K-**J**-I	(JxK)-(24x24)A (64x64)M	JC-KC-I-J-K	JC-KC-IR-JR-K-**J**-I
CHOL	K-J-I	(JxI)-(4x4)	JR-IR-K-**J**-I	(KxI)-(36x32)A (64x64)M	KC-IC-J-K-I	KC-IC-JR-IR-K-**J**-I
QR	J-K-I	(IxK)-(3x6)	IR-KR-J-**I**-K	(KxJ)-(12x20)A (54x60)M	KC-JC-I-K-J	KC-JC-IR-KR-J-**I**-K
SSYR2K	K-J-I	(KxJ)-(4x4)	KR-JR-I-**K**-J	(KxI)-(12x24)A (40x60)M	KC-IC-J-K-I	KC-IC-JR-KR-I-**K**-J
SSYRK	K-J-I	(KxJ)-(6x3)	KR-JR-I-**K**-J	(KxI)-(36x32)A (66x64)M	KC-IC-J-K-I	KC-IC-JR-KR-I-**K**-J
SSYMM	J-I-K	(JxI)-(2x4)	JR-IR-K-**J**-I	(JxK)-(8x28)A (40x60)M	JC-KC-I-J-K	JC-KC-IR-JR-K-**J**-I
STRMM	K-J-I	(JxI)-(4x4)	JR-IR-K-**J**-I	(JxK)-(24x24)A (64x64)M	JC-KC-I-J-K	JC-KC-IR-JR-K-**J**-I
STRSM	K-J-I	(JxI)-(4x4)	JR-IR-K-**J**-I	(JxK)-(24x24)A (64x64)M	JC-KC-I-J-K	JC-KC-IR-JR-K-**J**-I

Table 5.3: Tiled dimensions, loop order and tile sizes selected at each level for each benchmark program. Cache tile sizes are given both for the ALPHA (A) and for the MIPS (M).

Program	Main Loop Body
MMtri	C(I,J)=C(I,J)+A(I,K)*B(K,J)
LU	A(I,J)=A(I,J)-A(I,K)*A(K,J)
CHOL	A(I,J)=A(I,J)-A(I,K)*A(J,K)
QR	T1=A(J-I-1,K) T2=A(J-I,K) A(J-I-1,K)=C(I,J)*T1-S(I,J)*T2 A(J-I,K)=S(I,J)*T1+C(I,J)*T2
SSYR2K	C(I,J)=C(I,J)+B(J,K)*A(I,K)+A(J,K)*B(I,K)
SSYRK	C(I,J)=C(I,J)+A(J,K)*A(I,K)
SSYMM	C(K,J)=C(K,J)+A(K,I)*B(I,J) C(I,J)=C(I,J)+A(K,I)*B(K,J)
STRMM	B(I,J)=B(I,J)+A(I,K)*B(K,J)
STRSM	B(I,J)=B(I,J)-A(I,K)*B(K,J)

Table 5.4: Main loop body of the original code for each benchmark program.

5.4 PERFORMANCE EVALUATION

In this section we present quantitative data showing the effects of tiling for two different memory levels, the cache and register levels. In particular, for the four program versions mentioned previously, we will show (1) the performance that can be achieved, (2) the number of load/store instructions executed, (3) the number of cache misses and (4) the number of TLB misses. All these measurements will be presented both for the ALPHA 21164 and the MIPS R10000.

Performance

We start showing the performance obtained on both processors by the four versions of our benchmark programs. We will use the MFLOP/s metric as our indicator of performance and we recall that operations such as DIV and SQRT, that appear in statements outside the innermost loop in programs LU, CHOL, QR and STRSM (see Appendix C), are counted as only one operation.

Figures 5.6 and 5.7 present results for small-to-medium problem sizes[7] (from 10 to 100) and Figures 5.8 and 5.9 present results for medium-to-large problem sizes (from 100 to 1500) on the ALPHA and MIPS processors, respectively.

Let's first point out some program's behaviors for small-to-medium problem sizes where data does fit into the cache level. As it can be seen on both processors, the behavior of the four versions in all programs is similar. First, TRL and TCRL achieve more or less the same level of performance. Thus, the extra loop overhead of TCRL with respect to TRL has not a significant effect on processor performance. Only in program SSYRK and on the MIPS processor (Fig. 5.7), there is a noticeable difference. However, this performance difference is not due to the extra loop overhead of TCRL. We have inspected the assembler codes of TRL and TCRL generated by the MIPS compiler and we have found that it generates different code for the main loop body[8] for each version. While the main loop body in TRL was scheduled in 25 cycles, the main loop body in TCRL was scheduled in 26 cycles. We note that the main loop body is exactly the same in both versions.

Second, the performance of TRL and TCRL is always much better than the performance of TCL and UnOpt. This performance improvement achieved when tiling includes the register level is due to two reasons: 1) tiling for the register level always achieves a certain amount of ILP in the loop body (this is specially important in modern microprocessors capable to issue multiple instructions per cycle) and 2) the number of load/store instructions is significantly reduced (leading to a reduction of the critical path length).

7.A matrix size (or problem size) of N means that we are using matrices of N by N elements.
8.We refer as the main loop body to the loop body of the loop nest that traverses non-boundary tiles, that is, the loop nest where the two innermost loops are fully unrolled. This is also the most executed loop nest.

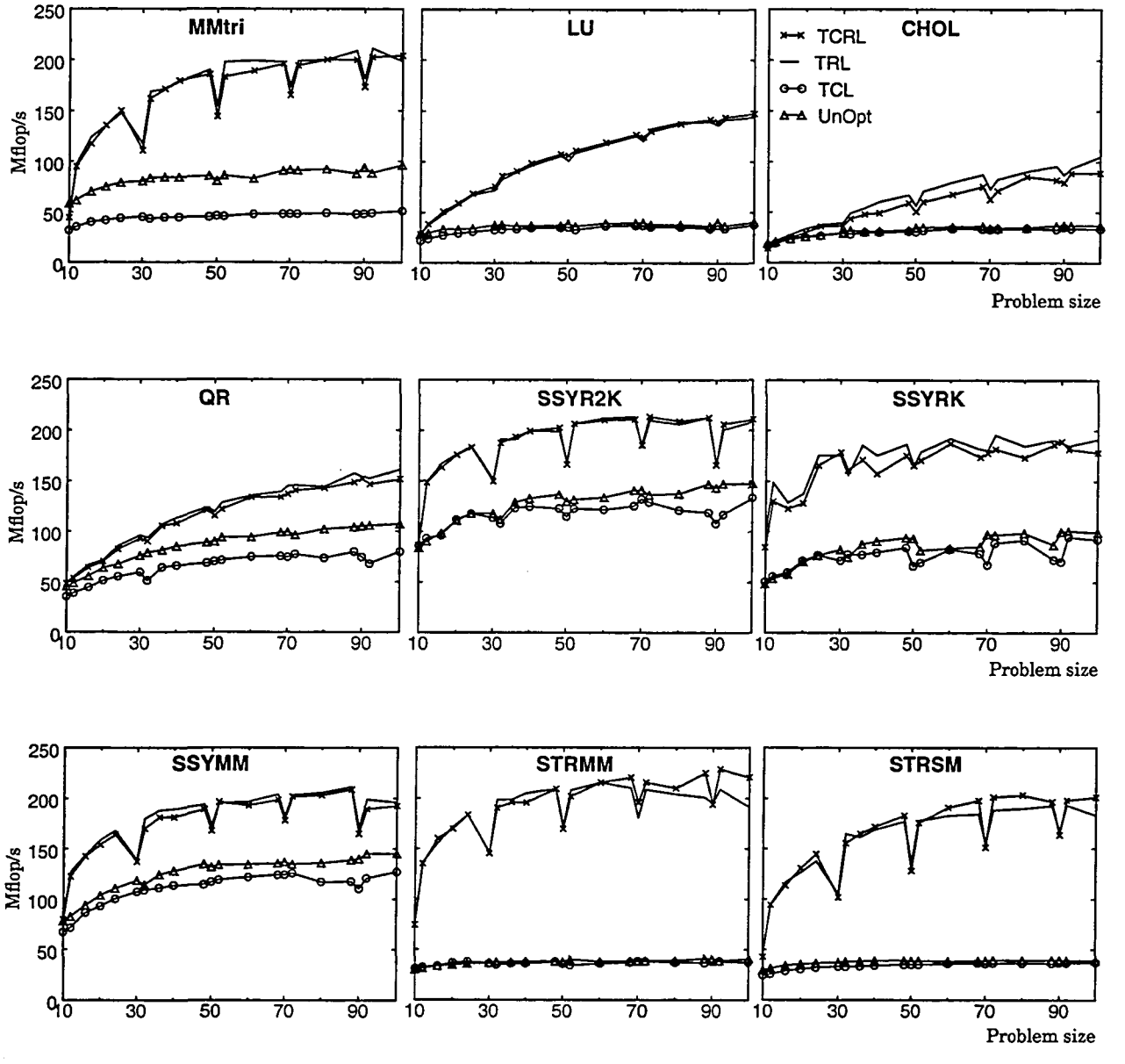

Figure 5.6: Performance obtained on the ALPHA 21164 processor by the UnOpt, TCL, TRL and TCRL versions for our benchmark programs, varying the problem size from 10 to 100.

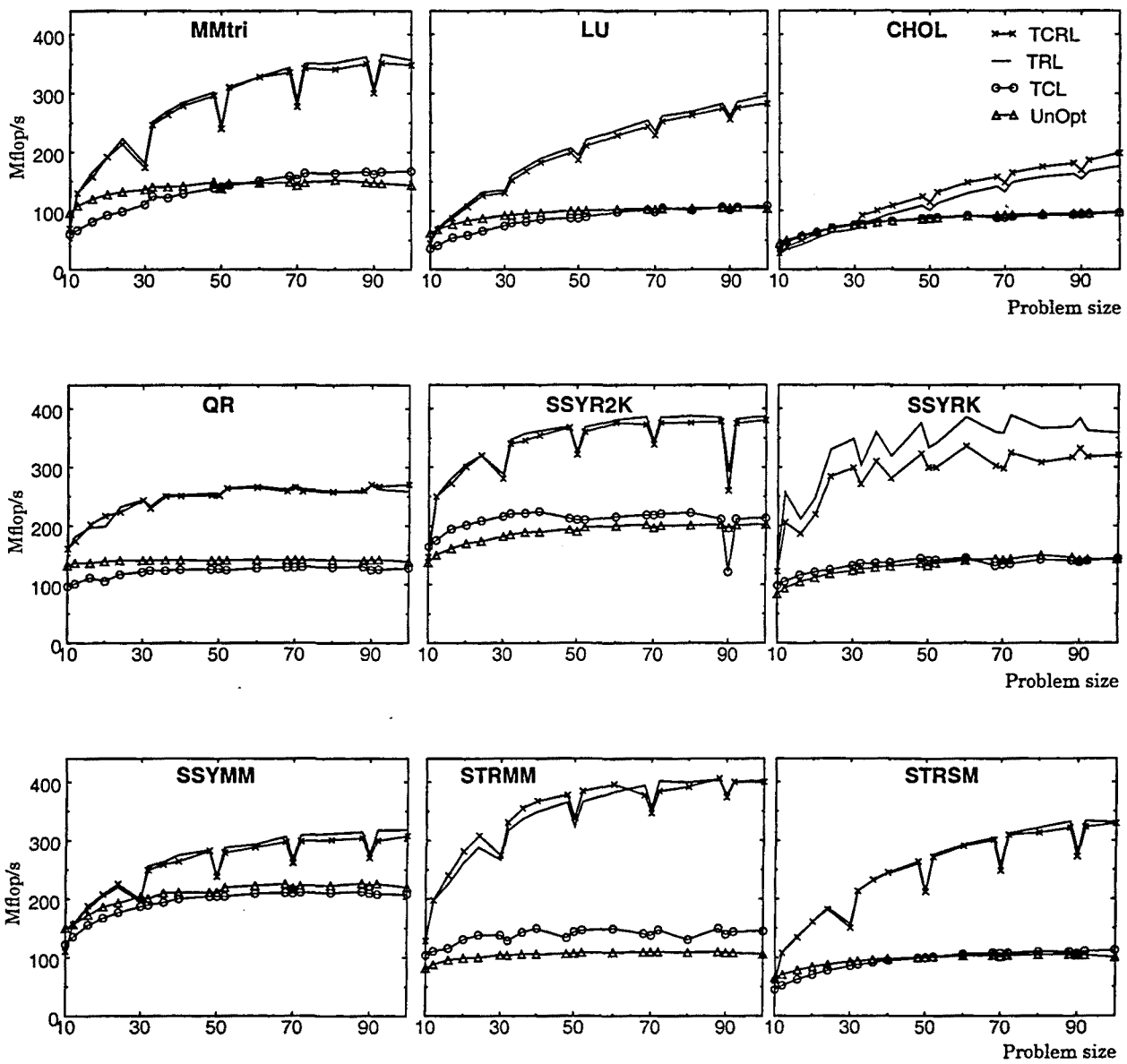

Figure 5.7: Performance obtained on the MIPS R10000 processor by the UnOpt, TCL, TRL and TCRL versions for our benchmark programs, varying the problem size from 10 to 100.

Finally, we want to note that, for some programs, there is a performance difference between TCL and UnOpt. Moreover, for some programs TCL performs better (see STRMM in Fig. 5.7) and for others UnOpt performs better (see MMtri and QR in Fig. 5.6). As already mentioned, for the UnOpt version we select the loop order that achieves, in average, the best performance and for the TCL version we use the same cache tiling parameters as for TCRL. Thus, these two versions of the programs have different loops as the innermost loop (see Table 5.3); the compiler generates different codes for the main loop body and, therefore, obtains different levels of performance. Anyway, our goal is not to compare TCL against UnOpt, but rather show that it is very important to exploit the register level in modern microprocessors.

Let's now see the program's behaviors for medium-to-large problem sizes (Figures 5.8 and 5.9) where data do not fit in the cache level. As it can be seen on both processors, the behavior of the four versions in all programs is similar: the performance obtained by the TRL and UnOpt versions decreases with large problem sizes, while the TCL and TCRL versions maintain the same level of performance for medium-to-large matrix sizes.

Tiling for the cache level is effective for reducing the capacity cache miss rate. Thus, for great matrix sizes that do not fit at the cache level it achieves the same performance as for smaller sizes. We can see that the TCL and TCRL versions achieve a stable performance level, mostly independent of matrix size, while the Unopt and TRL versions clearly decrease their performance when matrix size is increased. However, the TRL version achieves better performance than the TCL version even for medium matrix sizes that do not fit inside the first level cache. This is due to two reasons: (1) the typical bandwidth provided between the register file and the functional units in current superscalar microprocessors is three or four times higher than the bandwidth provided by the first level cache. If the register level is properly exploited, then the number of first level cache ports does not harm excessively processor performance and (2) tiling for the register level reduces memory instructions significantly, reducing therefore total data memory traffic. This reduction of total memory traffic helps reducing cache misses.

It can also be seen, that on the MIPS processor, the performance of TRL and UnOpt begins to decrease at much larger problem sizes than on the ALPHA processor, because the MIPS processor has a first level cache four times bigger than the ALPHA processor (see Table 5.2). Moreover, MIPS's cache is two way associative while ALPHA's cache is direct mapped. It is well known that having a set-associative cache reduces the frequency of interferences, thus reducing conflict misses [31][42][80].

Summarizing, the results show that it is much more important to exploit the register level than to exploit just the cache level. Nevertheless, best performance is achieved by tiling for both levels simultaneously, because tiling for both levels achieves 4 goals: (1) ILP is improved, (2) the number of load/stores instructions is significantly reduced, (3) capacity cache misses are moderated and, at last, (4) TLB misses are also moderated.

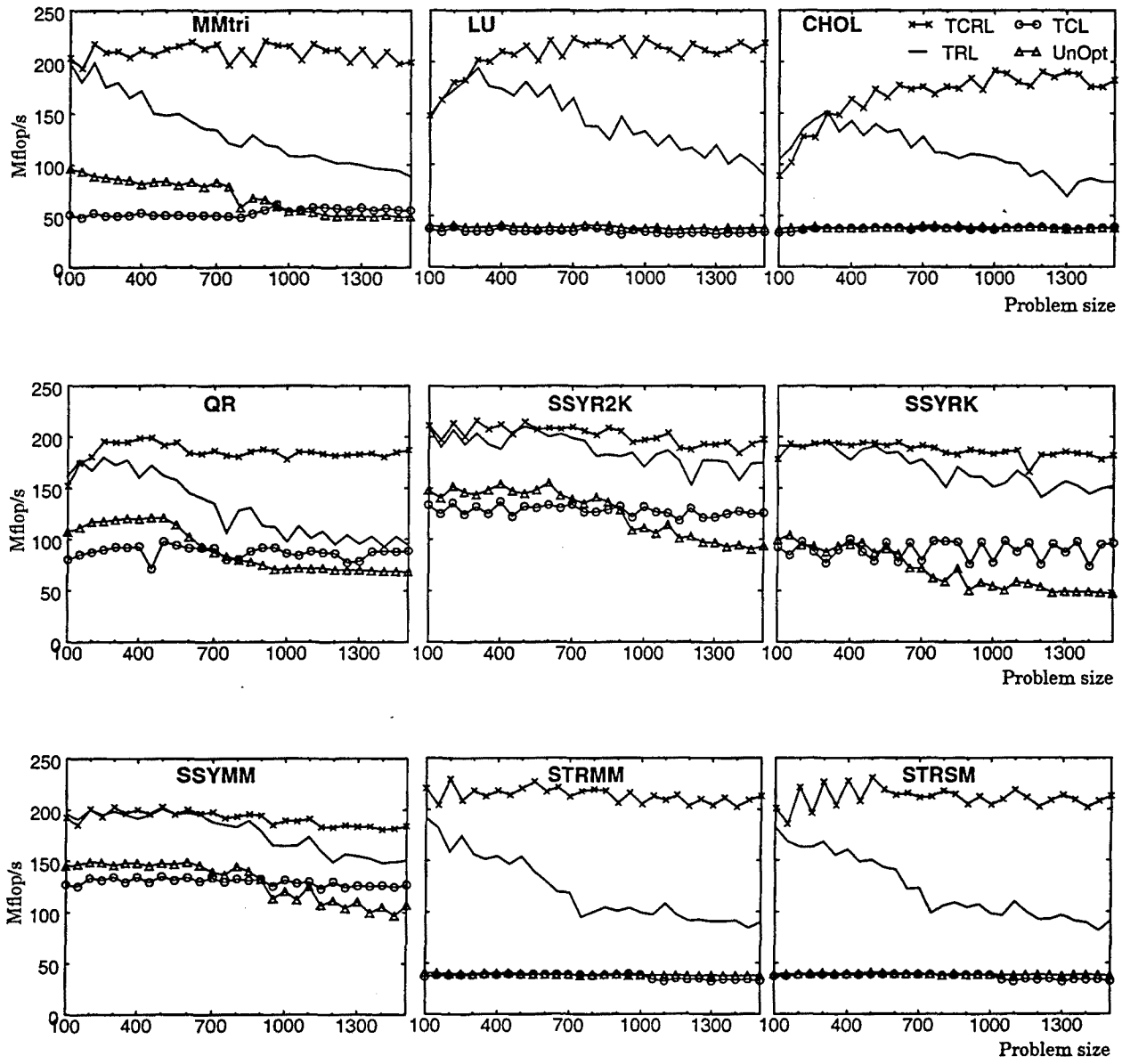

Figure 5.8: Performance obtained on the ALPHA 21164 processor by the UnOpt, TCL, TRL and TCRL versions for our benchmark programs, varying the problem size from 100 to 1500.

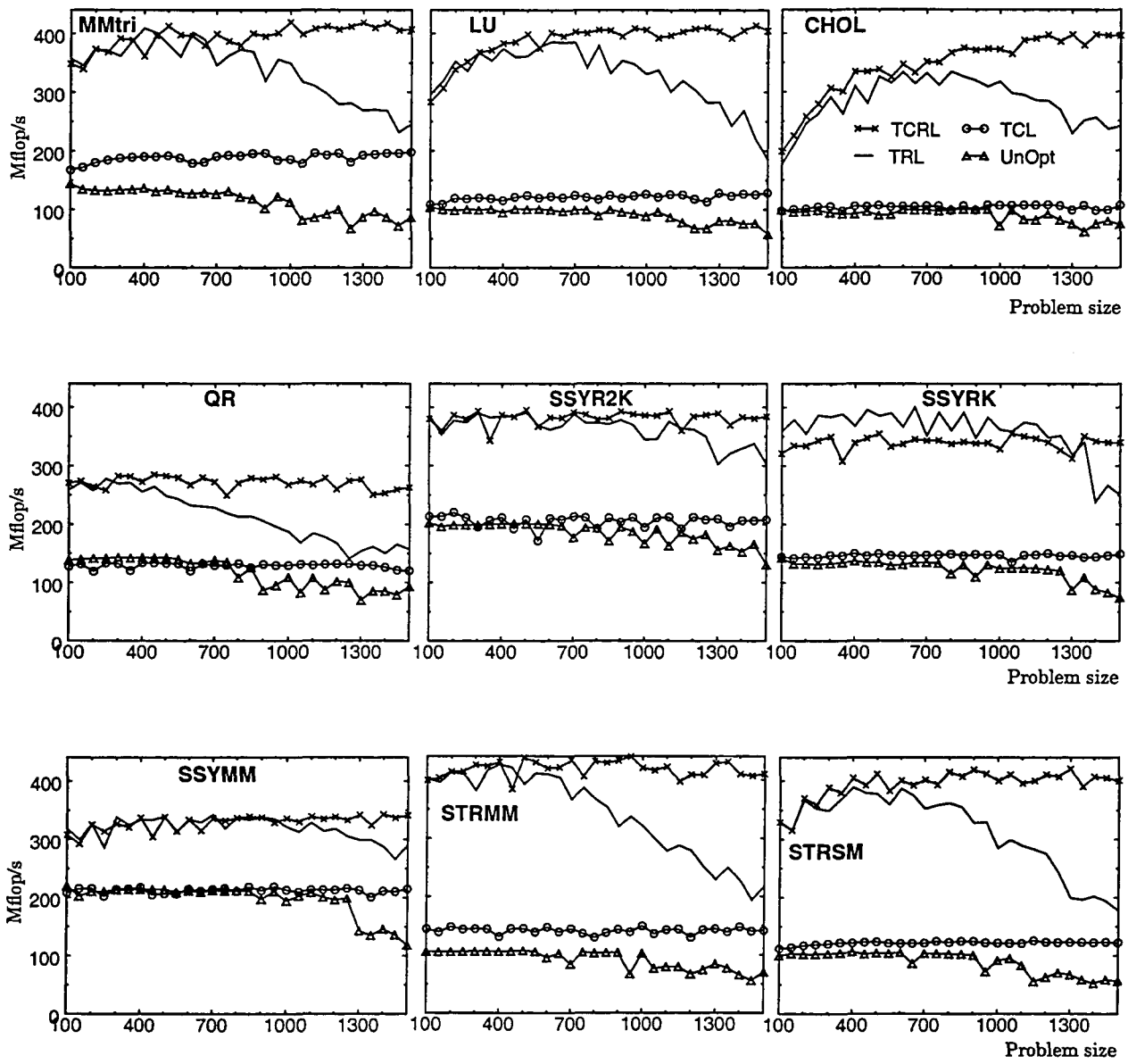

Figure 5.9: Performance obtained on the MIPS R10000 processor by the UnOpt, TCL, TRL and TCRL versions for our benchmark programs, varying the problem size from 100 to 1500.

Finally, we also want to note that tiling for the register level is beneficial on both in-order (ALPHA) and out-of-order (MIPS) processors. However, despite both processors having similar peak performance, the MIPS processor does generally outperform the ALPHA processor. We attribute this to three main advantages of the R10000: out-of-order execution, larger L1 cache and larger TLB[9].

Load/Store Instructions

As already mentioned, an important benefit of tiling for the register level is a reduction of the number of load/store instructions executed. Let's now see how tiling for the register level reduces the number of load/store instructions. Table 5.5 shows the absolute number of load/store instructions (in millions) performed by the UnOpt, TCL, TRL and TCRL versions with a problem size of 700 and the reduction percentage of TRL with respect to UnOpt, on the ALPHA and MIPS processors. We collected these numbers by instrumenting each executable with the ATOM tool [36] on the ALPHA processor and with the MINT3 tool [120] on the MIPS processor.

We can see that in all programs and on both processors the absolute number of load/store instructions in TRL and TCRL is significantly reduced with respect to UnOpt and TCL. As expected, TCRL performs slightly more load/store instructions than TRL, because, in TCRL, the non-tiled loop at the register level is tiled at the cache level. Thus, this non-tiled loop performs less iterations in TCRL than in TRL and, therefore, fewer loads and stores are removed. The reduction percentage of TRL with respect to UnOpt varies from a 50% in QR up to 88% in STRMM and STRSM on the ALPHA processor and from 31% up to 87% (in the same programs) on the MIPS processor. Note that the three

Program	ALPHA 21164					MIPS R10000				
	UnOpt	TCL	TRL	TCRL	% Red (TRL-UnOpt)	UnOpt	TCL	TRL	TCRL	% Red (TRL-UnOpt)
MMtri	402	431	61	71	85%	345	234	62	69	82%
LU	515	515	64	73	87%	458	346	64	71	86%
CHOL	258	259	35	40	86%	230	230	36	37	84%
QR	610	623	304	322	50%	577	577	400	404	31%
SSYR2K	818	831	294	313	64%	689	695	306	329	55%
SSYRK	602	607	118	124	80%	517	519	137	142	73%
SSYMM	816	842	221	239	73%	689	701	235	256	66%
STRMM	771	770	89	104	88%	686	518	89	95	87%
STRSM	772	773	90	104	88%	687	526	90	98	87%

Table 5.5: Total number of load/store instructions (in millions) performed by each version (UnOpt, TCL, TRL and TCRL) on the ALPHA 21164 and the MIPS R10000 processors and the reduction percentage of TRL with respect to UnOpt. In all cases, the problem size is equal to 700.

9.Both the ALPHA and MIPS processors have the same number of TLB entries, however, the page size on the MIPS processor is 4 times lager than the ALPHA's page size.

programs where the reduction is smaller (QR, SSYR2K and SSYMM) are also the three programs where the performance difference between UnOpt and TRL is smaller (see Figures 5.8 and 5.9).

Another interesting point to note is that the reduction percentage of TRL with respect to UnOpt is smaller on the MIPS processor than on the ALPHA processor. Note that the UnOpt version, as well as TCL, perform less memory instructions on the MIPS processor than on the ALPHA processor. This is because the MIPS compiler usually unrolls the innermost loop with an unrolling factor greater than the unrolling factor used by the ALPHA compiler. Increasing the unrolling factor reduces the number of memory instructions, if the unrolled loop carries reuse. Moreover, the MIPS compiler applies scalar replacement to almost all programs (thus removing redundant loads and stores in the loop body), while the ALPHA compiler was able to apply scalar replacement to fewer programs.

We also want to note that, for some programs (QR, SSYR2K, SSYRK and SSYMM), the TRL and TCRL versions execute a few more load/store instructions on the MIPS processor than on the ALPHA processor, just the opposite of what happened with the UnOpt and TCL versions. In these four programs, the tile size selected for the register level is quite large, forcing the compiler to generate spill code. Note that the TRL and TCRL versions of programs QR, SSYR2K, SSYRK and SSYMM require more registers than do the remaining programs (see Tables 5.3 and 5.4). As mentioned in Chapter 3, using large tile sizes increases register pressure and, therefore, spill code can be generated in the main loop body. However, if this increase of memory instructions due to spill-loads/stores is less than the reduction of memory instructions due to register tiling, then it is preferable to use bigger tile sizes although they generate a certain amount of spill code than to use smaller tile sizes. We have inspected the assembler code of these four programs on both processors and we have seen that the MIPS compiler generates more spill code in the main loop body than the ALPHA compiler and, thus, the number of loads/stores instructions on the MIPS processor increases.

Finally, we want to indicate that tiling for the register level not only reduces load/store instructions but also other types of instructions (such as integer arithmetic, logical, shift and branch instructions) used in the computation of effective addresses and in loop control. On one hand, the instructions related to the computation of effective addresses are reduced because fewer memory instructions are executed. On the other hand, the instructions that perform loop control are reduced because tiling for the register level uses "large" unrolling factors. With a tile size of 4 by 4, for example, the unrolled loop body has 16 instances of the original loop body. Nevertheless, if register tiling is not performed (and assuming that compilers usually unroll the innermost loop by a factor of 4), the unrolled loop body will only have 4 instances of the original loop body. Thus, in the main loop body, register tiling reduces loop control instructions by a factor of 4 with respect to the not tiled code. Of course, register tiling has a loop overhead due to the iteration space tiling phase. However, the number of instructions introduced by this loop overhead is less than the instructions saved by unrolling with a large factor.

We measured for all programs the total number of executed instructions and subtracted both memory and floating-point instructions. We collected these numbers only for the UnOpt and TRL versions and we used a problem size of 700. We found that TRL performs around 75%-85% less instructions than UnOpt for all programs and on both processors.

Cache misses

Let's now see how cache misses are reduced when tiling includes the cache level. Table 5.6 shows the absolute number of first level cache misses (in millions) performed by the UnOpt, TCL, TRL and TCRL versions of each program on the ALPHA and MIPS processors. We have used, again, a matrix size of 700 for the measurements. We present absolute number of misses rather than cache miss rate because the number of load/store instructions performed by each version is different. However, cache miss rates can be computed using Tables 5.5 and 5.6 and results shows that, as expected, the cache miss rate in TCL is reduced with respect to Unopt, and the miss rate of TCRL is reduced with respect to TRL. However, the cache miss rate of TRL and TCRL, in almost all programs, is greater than the cache miss rate of UnOpt, because they (TRL and TCRL) execute less load/store instructions.

In Table 5.6, we can see that the TRL and TCL versions always have fewer cache misses than the UnOpt version but the TCRL version is the one with fewer overall misses. Although the cache level is not being exploited in TRL, it reduces overall cache misses (with respect to UnOpt) because, as already mentioned, it reduces significantly total memory traffic.

Note that there are programs (all programs on the MIPS processor, except SSYR2K, and programs CHOL, QR and SSYMM on the ALPHA processors) where the TCL version performs fewer absolute number of cache misses than the TRL version. However, the TCL version always achieves

Program	ALPHA 21164				MIPS R10000			
	UnOpt	TCL	TRL	TCRL	UnOpt	TCL	TRL	TCRL
MMtri	20.1	17.3	10.4	3.8	14.8	2.8	4.3	1.2
LU	21.5	15.2	14.3	4.9	14.5	2.3	6.5	1.3
CHOL	10.2	5.2	9.4	2.9	7.4	0.8	4.8	0.5
QR	79.9	27.5	32.1	16.6	38.9	12.4	23.6	7.1
SSYR2K	38.8	31.6	22.1	14.7	23.4	9.7	9.3	5.3
SSYRK	30.3	13.6	12.7	6.9	22.6	3.3	4.5	2.5
SSYMM	38.6	20.1	22.6	11.8	23.4	9.8	13.0	5.2
STRMM	30.6	24.9	16.7	6.1	22.6	4.6	7.1	2.4
STRSM	31.2	24.7	16.8	5.7	22.6	4.5	7.2	2.4

Table 5.6: Total number of first level cache misses (in millions) performed by the four versions (UnOpt, TCL, TRL and TCRL) for each program on the ALPHA 21164 and the MIPS R10000 processors. In all cases, the problem size is equal to 700.

less performance than the TRL version as seen in Figures 5.8 and 5.9. It is much more important to reduce load/store instructions and improve ILP by exploiting the register level than to reduce only the cache misses by exploiting the cache level. Nevertheless, best performance is achieved by tiling for both levels simultaneously (TCRL), because it achieves the benefits of both individual levels. As can be seen in Table 5.6, TCRL is the one with fewer overall misses.

Finally, we want to note that the absolute number of cache misses on the MIPS processor is always smaller than the overall misses on the ALPHA processor, because, as already mentioned, the cache of the MIPS is four times bigger than the ALPHA's cache and MIPS's cache is two-way associative while ALPHA's cache is direct mapped.

TLB misses

We have seen so far that tiling for the register level decreases significantly the number of memory operations and also reduces the overall number of cache misses. However, tiling only for the register level can reduce the amount of spatial reuse that can be exploited, heavily increasing the overall TLB misses.

Table 5.7 shows the absolute number of TLB misses (in thousands) performed by the UnOpt, TCL, TRL and TCRL versions of each program on the ALPHA and MIPS processors. In this case, for the measures we have used a problem size of 700 on the ALPHA processor and of 1000 on the MIPS processor. We use a higher problem size on the MIPS processors because a matrix of 700x700 elements

Program	ALPHA 21164					MIPS R10000				
	UnOpt	TCL	TRL	TCRL	% Incr (TRL-UnOpt)	UnOpt	TCL	TRL	TCRL	% Incr (TRL-UnOpt)
MMtri	166	5	2123	5	1179%	117	1.6	1320	1.6	1028%
LU	247	172	2246	172	809%	169	124	1345	124	696%
CHOL	79	16	1062	16	1244%	46	3.3	610	3.3	1226%
QR	168	30	8848	30	5167%	117	5.5	8840	5.5	7455%
SSYR2K	672	707	546	707	-19%	491	390	398	390	-19%
SSYRK	503	389	364	389	-38%	369	259	266	259	-28%
SSYMM	672	767	588	767	-14%	489	390	428	390	-12%
STRMM	503	343	3794	343	654%	368	247	3191	247	767%
STRSM	504	343	3822	343	658%	369	247	3103	247	741%

Table 5.7: Total number of TLB misses (in thousands) performed by each version (UnOpt, TCL, TRL and TCRL) on the ALPHA 21164 and the MIPS R10000 processors and the increment percentage of TRL with respect to UnOpt. The problem size is equal to 700 on the ALPHA processor and to 1000 on the MIPS processor.

fits entirely in the TLB[10] (recall from Table 5.2 that each TLB entry on the MIPS processor maps two consecutive pages of 16KB) and the goal of this measurements is to show how tiling for the register level increases TLB misses if there are not enough TLB entries.

We can see that TRL versions heavily increases the number of TLB misses (between 654% and 5167% on the ALPHA processor and between 696% and 7455% on the MIPS processor) in all programs, except in those three (SSYR2K, SSYRK and SSYMM) in which no array reference is traversed in row major order[11] by the non-tiled loop at the register level (see Tables 5.3 and 5.4). In these cases the TLB misses are reduced (with respect to the UnOpt version) between 14% and 38% on the ALPHA processor and between 12% and 28% on the MIPS processor. We note that in program QR, where this increase in TLB misses is top-heavy, there are two array references accessed in row major order, while in MMtri, LU, CHOL, STRMM and STRSM there is only one.

As shown in Section 5.2.1, if there is at least one array reference that is traversed in row major order by the non-tiled loop at the register level, spatial locality is not being exploited for these references, and this fact makes TLB misses increase considerably. Nevertheless, as we have seen previously, despite this increase in TLB misses high performance can be achieved by tiling only for the register level.

On the other hand, as shown in Section 5.2.2, tiling only for the cache level (TCL version) can also increase the overall TLB misses (with respect to the non-tiled code), if there is at least one array reference that is traversed in row major order by the non-tiled loop. In programs SSYR2K, SSYRK and SSYMM there is one array reference traversed in row major order by the non-tiled loop of the TCL versions. In these versions, the overall TLB misses will increase (with respect to the UnOpt versions) if the factor $N/(B_1*B_2)$ is greater than 1, where B_1 and B_2 are the tile sizes in each tiled dimension and NxN is the size of the matrices. Using the tile sizes presented in Table 5.3 and being N=700 on the ALPHA processor and N=1000 on the MIPS processor, it can be seen that this factor is greater than one only for the SSYR2K and SSYMM programs on the ALPHA processor. Thus, as can be seen in Table 5.7, for these two programs and only on the ALPHA processor, the overall TLB misses of TCL increases with respect to the UnOpt version. However, because the tile size for the cache level is much larger than the tile size for the register level, this increase in overall TLB misses is not as significant as at the register level.

Finally, by tiling for both levels (TCRL version) the TLB misses are again significantly reduced, performing in all cases the same TLB misses as the TCL version. Tiling only for the register level can produce excessive TLB misses. However, as shown in Section 5.2.3, when tiling also for the cache level using MOB forms, all directions of the iteration space are bounded from the point of view of the register level and thus, the number of TLB misses is moderated.

10. We consider a matrix to fit "in the TLB" if all virtual to physical address translations for all matrix's virtual pages fit in TLB.
11. Recall that matrices are stored in column major order.

5.5 HAND vs. AUTOMATICALLY-OPTIMIZED CODES

In this section, we will compare our automatic-optimized codes against hand-optimized codes and we will show how our techniques allow automatic-optimized codes to achieve the same performance level as hand-optimized codes, even when dealing with complex numerical codes with non-rectangular iteration spaces.

We will use as benchmark five programs extracted from the BLAS3 library [38][39]. The automatic-optimized codes were produced using our own developed tool that implements the methods proposed in Chapter 3 and Chapter 4 and, as hand-optimized codes, we have used two different libraries: the BLAS3 library provided by the manufacturers and the RISC-BLAS library proposed in [34][35]. The RISC-BLAS library provides an efficient and portable implementation of the Level 3 BLAS for RISC processors. This library express the BLAS as a sequence of matrix-matrix multiplications and operation involved triangular blocks. All the codes in the RISC-BLAS are written in Fortran, use loop unrolling, loop tiling and data copying [116] to improve performance and are specifically tuned for RISC processors. We will first describe the evaluation process used in this section and then we present the performance results.

5.5.1 Evaluation Process

As just mentioned, we have used as benchmark programs 5 linear algebra algorithms extracted from the BLAS3 library [38][39]. In particular, we have used the SSYR2K, SSYRK, SSYMM, STRMM and STRSM algorithms that have non-rectangular, 3-dimensional iterations spaces. Table 5.1 on page 194 describes each of them.

All our measurements were taken, again, on a uniprocessor system with an ALPHA 21164 processor [15] and on a single R10000 processor [130] of a multiprocessor system (SGI Origin 2000 [81]). The two different architectures are shortly described in Table 5.2 on page 195.

We evaluate four different versions of each program: one uses the original code as proposed by [38][39] with no restructuring transformation[12] (ORI-blas); the second one uses the manufacturer-supplied BLAS3 library to perform the operation (VENDOR-blas); the third one calls the RISC-BLAS library[13] (RISC-blas); and the fourth one is the code after tiling for both cache and register levels using our own developed tool (TCRL).

After generating the different versions for each program, we used the standard Fortran 77 compiler[14] to generate the final executables. In general, the F77 compiler was used with the highest scalar optimization level (-O5 on the ALPHA and -O3 on the MIPS) turned on. However, in

12.The original codes are available at http://www.netlib.org/blas.
13.The RISC-BLAS library is available using ftp anonymous at ftp.enseeiht.fr. It is located in pub/numerique/BLAS/RISC.
14.Version 4.1 of Digital Fortran on the ALPHA processor and version 7.2.1 of MIPSpro Compiler on the MIPS processor.

Program	Operation	Routine Call
SSYR2K	C = BETA*C + ALPHA*A*B^T + ALPHA*B*A^T C is symmetric and only the upper triangular part is to be referenced	SSYR2K ('U', 'N', N, N, ALPHA, A, N, B, N, BETA, C, N)
SSYRK	C = BETA*C + ALPHA*A*A^T C is symmetric and only the lower triangular part is to be referenced	SSYRK ('L', 'N', N, N, ALPHA, A, N, BETA, C, N)
SSYMM	C = BETA*C + ALPHA*A*B A is symmetric and only the upper triangular part is to be referenced	SSYMM ('L', 'U', N, N, ALPHA, A, N, B, N, BETA, C, N)
STRMM	B = ALPHA*A*B A is an upper unit triangular matrix	STRMM ('L','U', 'N', 'U', N, N, ALPHA, A, N, B, N)
STRSM	B = ALPHA*inv(A)*B A is a lower non-unit triangular matrix	STRSM ('L','L', 'N', 'N', N, N, ALPHA, A, N, B, N)

Table 5.8: Operation performed in the TCRL versions and the routine calls used in the ORI-blas, VENDOR-blas and RISC-blas versions for each benchmark program.

the TCRL version of some programs, the compiler generates a more efficient code using the -O4 (on the ALPHA) and -O2 (on the MIPS) flags. In these cases, we have used these later flags. In addition, the RISC-blas version was linked with the RISC-BLAS library tuned to each target architecture [34] and the VENDOR-blas version was linked with the vendor-supplied BLAS3 library, using -lblas on the R10000 and -ldxml on the ALPHA processor[15].

To have a fair comparison with the BLAS3 libraries, the TCRL version performs exactly the same operations as the BLAS3 routines. Table 5.8 shows the operation performed in the TCRL version of each program, where ALPHA and BETA are scalars, A, B and C are N by N matrices and A^T and B^T denote the transpose of A and B, respectively. Table 5.8 also shows the routine calls used in the ORI-blas, VENDOR-blas and RISC-blas versions. In the next section, we will present performance results for only these particular routine calls. However, we want to mention that similar results are achieved if different parameters were used. Finally, we note that in Section 5.4 of this chapter and in Section 3.5.2 of Chapter 3, the different versions of the programs did not perform exactly the same operations as the BLAS3 routines. In particular, they did not perform the operations where the scalar variables ALPHA and BETA were involved (see Table 5.4 and Appendix C).

As in the previous section, for the TCRL versions, we always tile, for each level, two dimensions of the 3-dimensional iteration spaces [98] and we use MOB forms. We have chosen the tiling parameters for the cache and register level that provide best performance on the two different processors. Again, we want to stress that our goal is not finding the optimum tile sizes and shape for each level, but showing that automatic-optimized codes can achieve the same performance level as hand-optimized codes. Hence, we use the same tiles (size and shape) for different problem sizes and do not evaluate problem sizes that produce maximum self interference.

15.We have used Version 3.1.1 of the Silicon Graphics Scientific Mathematical Library, CHALLENGEComplib, that contains the extended Level 3 BLAS and Version 3.3 of the Digital Extended Math Library (DXML)

Program	Register Level		Cache Level		Loop Order
	tiled directions	tile sizes	tiled directions	tile sizes	
SSYR2K	(KxJ)	(4x4)A (3x4)M	(KxI)	(32x240)A (40x60)M	KC-IC-JR-KR-I-K-J
SSYRK	(KxJ)	(4x4)A (6x3)M	(KxI)	(32x240)A (66x64)M	KC-IC-JR-KR-I-K-J
SSYMM	(JxK)	(4x4)	(IxK)	(64x240)A (80x60)M	IC-KC-JR-KR-I-K-J
	(JxI)	(4x4)	(JxK)	(28x240)A (40x60)M	JC-KC-IR-JR-K-J-I
STRMM	(JxI)	(4x4)	(JxK)	(28x50)A (64x64)M	JC-KC-IR-JR-K-J-I
STRSM	(JxI)	(4x4)	(JxK)	(28x50)A (64x64)M	JC-KC-IR-JR-K-J-I

Table 5.9: Tiling parameters of the TCRL version for each benchmark program.

Table 5.9 summarizes for each program the tiling parameters selected for the cache and register levels in the TCRL versions. In particular, we show the tiled dimensions for the register level (column 2) and their corresponding tile sizes (column 3), the tiled dimensions for the cache level (column 4) and their corresponding tile sizes (column 5) and the final loop order (column 6). The tile sizes used on the ALPHA processor are labelled with "A" and the tile sizes used on the MIPS processor are labelled with "M". When no label is given, it indicates that we have used the same tile size on both processors. We note that, on the ALPHA processors, we have tuned the cache tiling parameters with respect to the second level of internal cache since our experiments show this is most efficient. Finally, we also want to note that on the SSYMM program we distributed the original loops to create two new loop nests; each new loop nest traverses one statement of the original loop body (Appendix C shows these two statements[16]). In Table 5.9 we present the tiling parameters for each of the two loop nests in the SSYMM program.

5.5.2 Performance Results

In this section we present the performance obtained by the four versions of our benchmark programs on the two different architectures. Figures 5.10 and 5.11 show the performance obtained for problem sizes varied from 10 to 100 and from 100 up to 1500 on the ALPHA and MIPS processors, respectively. We use again the MFLOP/s metric as our indicator of performance.

16.In fact, the two statements in the loop body of the SSYMM BLAS3-routine are not exactly the same as those in Appendix C. Recall that this appendix does not contain the operations where the scalar variables ALPHA and BETA are involved.

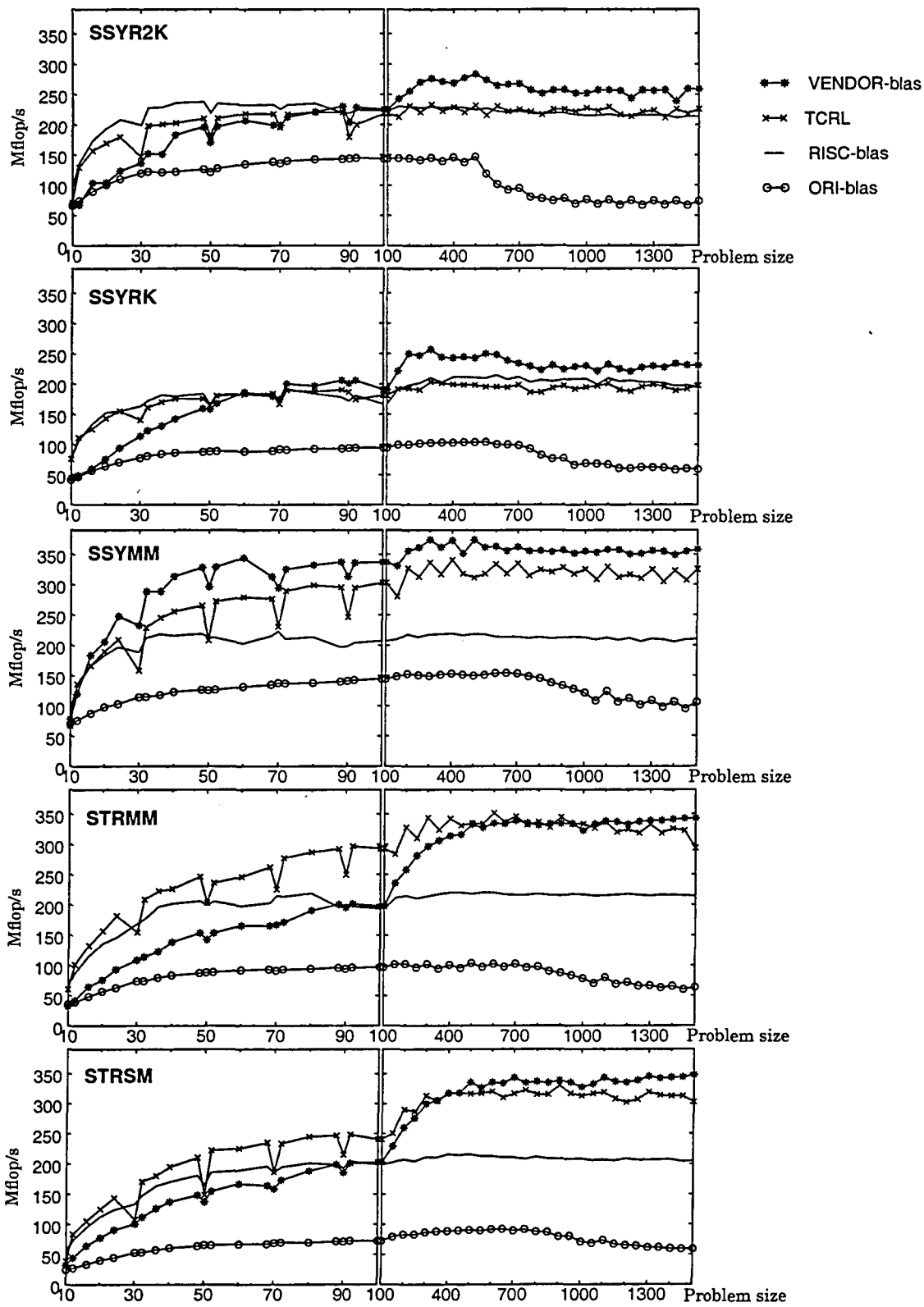

Figure 5.10: Performance obtained on the ALPHA 21164 processor by the ORI-blas, VENDOR-blas, RISC-blas and TCRL versions for our five benchmark programs, varying the problem size from 10 to 1500.

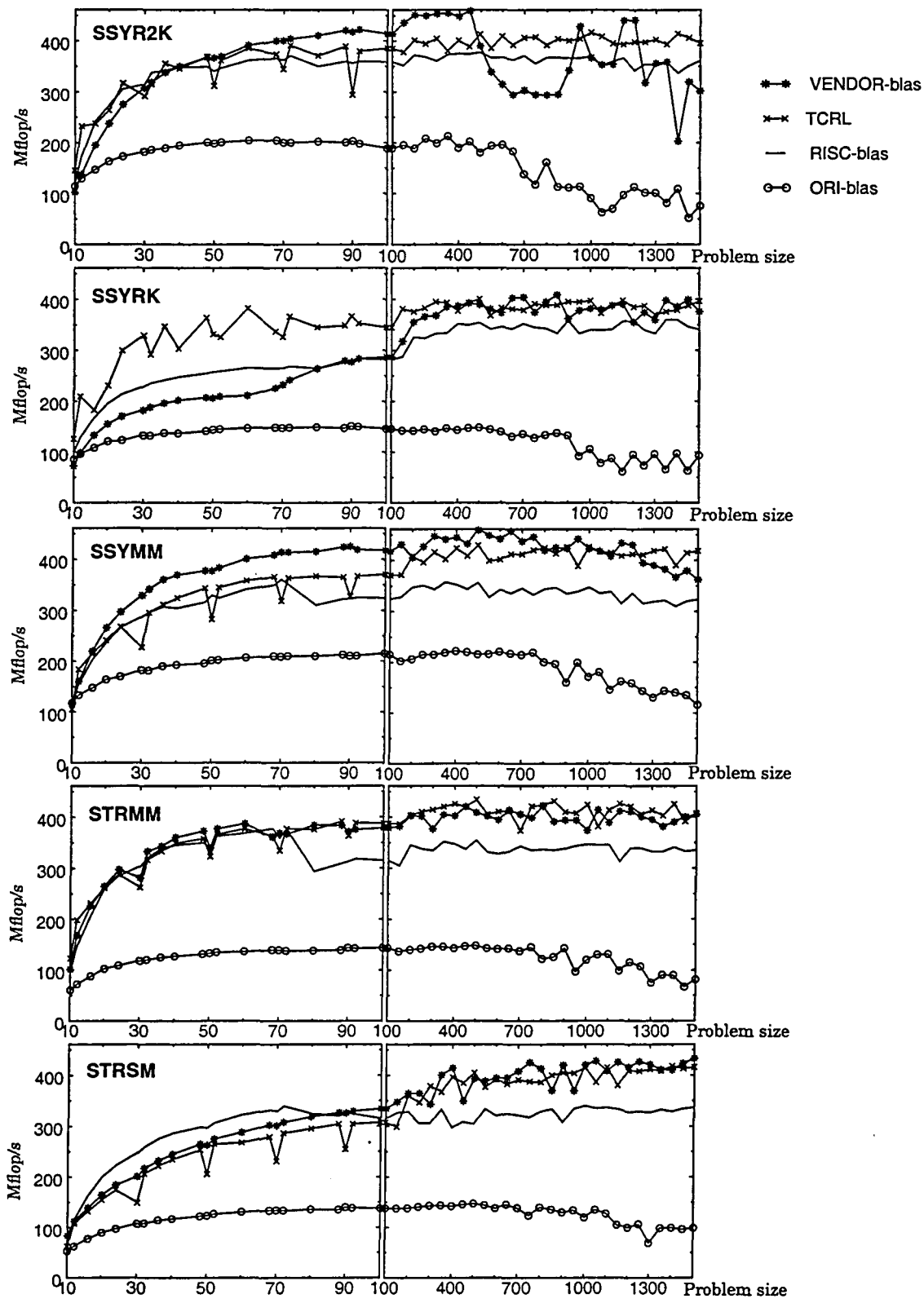

Figure 5.11: Performance obtained on the MIPS R10000 processor by the ORI-blas, VENDOR-blas, RISC-blas and TCRL versions for our five benchmark programs, varying the problem size from 10 to 1500.

On the ALPHA processor, it can be seen that, for small-medium problem sizes (from 10 to 100), TCRL and RISC-blas always perform better than the VENDOR-blas version, except for the SSYMM program where the VENDOR-blas version achieves better performance. Moreover, TCRL performs better or equal than RISC-blas (only for the SSYR2K, RISC-blas performs slightly better than TCRL).

By contrast, for medium-large problem sizes (from 100 to 1500), VENDOR-blas is always better than TCRL and RISC-blas, probably because better use is made of the memory hierarchy. In the TCRL version, we only perform two levels of tiling (registers and L2-cache) and we do not perform data copying [80][116]. We presume that the vendor-supplied BLAS3 library performs more levels of tiling (registers, L1-cache and L2-cache) and it also performs data copying. It is well known that data copying reduces conflict misses and allows to select bigger tile sizes, thus enhancing data locality. Of course, data copying may induce a large overhead that can outweigh the benefit. However, this is not usually the case for large problem sizes and for programs having a high degree of data reuse (as our benchmarks). Another interesting point to note is the inflection point that appears in all VENDOR-blas curves for a problem size equal to 100. Most probably, data copying is being used by the vendor code in problem sizes from 100 onwards.

Finally, we also want to note that, for medium-large problem sizes, TCRL is again better or equal than RISC-blas. The hand-optimized codes in the RISC-BLAS [35] use loop unrolling, loop tiling and, data copying. Moreover, the RISC-BLAS codes on the ALPHA processor are also tuned with respect to the second level cache. Although the RISC-BLAS codes are optimized for the second level cache, they do not always achieve as high performance as TCRL (see SSYMM, STRMM and STRSM). The reason is that they do not exploit the register level as well as TCRL and, if the register level is not properly exploited, then the number of ports of the first level cache bounds processor performance. Note that, precisely in these three programs and for small problem sizes that fit in the cache level, the RISC-blas version performs worst than TCRL.

On the MIPS processor, it can be seen that, for small-medium problem sizes, TCRL, VENDOR-blas and RISC-blas achieve more or less the same performance, except for the SSYMM program where, much like on the ALPHA processor, the VENDOR-blas version achieves better performance and, for the SSYRK program where TCRL clearly performs better than VENDOR-blas and RISC-blas. For the STRSM program, the RISC-blas version also performs slightly better than VENDOR-blas and TCRL. However, in this case, the performance improvement is due to a particular transformation performed on the RISC-BLAS STRSM code, that we also could have done in the TCRL version. In particular, this transformation consists in substituting a set of DIV operations (a very time consuming, non-pipelined operation), having all of them the same divisor, by one DIV and a set of MUL operations. These DIV operations, however, appear in statements outside the innermost loop (recall that STRSM is not perfectly nested). Therefore, the performance improvement caused by this optimizing transformation becomes less noticeable when the problem size increases.

Program	ALPHA 21164						MIPS R10000					
	small-medium sizes			medium-large sizes			small-medium sizes			medium-large sizes		
	VENDOR	TCRL	RISC	VENDOR	TCRL	RISC	VENDOR	TCRL	RISC	VENDOR	TCRL	RISC
SSYR2K	1.27	1.50	1.72	2.86	2.47	2.45	1.61	1.72	1.64	2.98	3.36	3.05
SSYRK	1.51	2.03	2.06	2.95	2.46	2.57	1.39	2.18	1.67	3.48	3.56	3.14
SSYMM	2.09	1.79	1.63	2.75	2.46	1.64	1.68	1.49	1.47	2.34	2.28	1.87
STRMM	1.47	2.56	2.23	3.76	3.90	2.58	2.50	2.56	2.33	3.38	3.48	2.85
STRSM	2.02	2.79	2.70	4.15	4.03	2.74	2.01	1.78	2.19	3.23	3.16	2.65

Table 5.10: Average speedups obtained by VENDOR-blas, TCRL and RISC-blas over the ORI-blas version for small-medium and medium-large problem sizes, on an ALPHA 21164 and a MIPS R10000 processor.

For medium-large problem sizes the behavior of all programs on the MIPS processor is, more or less, the same as on the ALPHA processor: the VENDOR-blas version is better or equal than TCRL and RISC-blas and the TCRL version is always better than RISC-blas. Finally, we also want to note the strange behavior of the SSYR2K VENDOR-blas version for some large problem sizes. We presume that the tiling parameters used by the vendor-supplied library incur in a significant increase in cache and TLB misses for certain problem sizes. In fact, we have measured the overall number of L1-cache, L2-cache and TLB misses performed by the VENDOR-blas and TCRL versions of the SSYR2K program for a problem size equal to 1000 and we have found that the VENDOR-blas version performs 7.5 times more TLB misses, 1.8 times more L2-cache misses and 3.6 times more L1-cache misses than TCRL.

To conclude this section, we have summarized the speedups obtained by VENDOR-blas, TCRL and RISC-blas versions over the ORI-blas version on both processors. For each program version the harmonic mean of the MFLOP/s obtained for different matrix sizes is computed (we used 21 different small-medium problem sizes, going from 10 to 100 and 29 different large-medium sizes, going from 100 to 1500). Then, we compute the speedup of the different versions over the ORI-blas version by dividing these harmonic means. Table 5.10 shows the average speedups for the small-medium and medium-large problem sizes on both processors.

Results shows that automatic-optimized codes (TCRL) can almost achieve the same performance as hand-optimized codes (VENDOR-blas and RISC-blas) and, in some cases (marked in dark in Table 5.10), TCRL achieves even better results than VENDOR-blas and RISC-blas. Thus, our techniques allow automatic-optimized codes to achieve the same performance level as hand-optimized codes, even when dealing with complex numerical codes having non-rectangular iteration spaces.

5.6 RELATED WORK

There is much work in the literature regarding loop tiling evaluation [27][31][65][70][80][92][98], but it has mostly focused on the cache level and little attention has been paid to the register level. Most of these works, [27][31][65][70][92], focus on determining the tiling parameters (which loops should be tiled, the tile sizes in each dimension and the relative order of the loops) that maximize data locality at the cache level. To this end, these works present analysis of the performance of several programs tiled for the cache level and study the effects of the tiling parameters on processor performance. However, to the best of our knowledge, there is no previous work evaluating the impact of tiling for the different memory levels (including the register level).

Lam et al. in [80] present an analysis of the performance of several programs tiled for the cache level and study the effects of different cache parameters, such as set associativity and line size, on processor performance. However, they do not study the effects of tiling for the different memory levels.

Navarro et al. in [98] evaluate the Multilevel Orthogonal Block forms analytically for an ideal memory system with M cache levels. Moreover, they also present experimental results of the MOB forms on two different workstations and indicate the most significant aspects that can affect the performance of the algorithms with respect to the ideal case. These aspects are (1) the need to perform register tiling, (2) the effect of the interferences in direct-mapped and set-associative caches, (3) the effect of TLB misses and (4) the effect on page faults. However, they only use the square matrix product in their experiments and do not perform an exhaustive study of all these aspects.

There has been also much work discussing the performance of blocked BLAS3 linear algebra routines [38][39] and providing new library routines. For example, Gallivan et al. in [48] discussed the performance of blocked BLAS3 linear algebra routines on an Alliant FX/8 and found that blocked versions generated twice the Mflop/s for some problem sizes. Samukawa in [109] proposed two types of matrix-matrix operation routines to be included in the BLAS standard which can be used as tools of various band and skyline matrix factorizations. Quintana-Orti at al. in [104] developed a new block variant of the QR factorization with column pivoting which allows the use of Level 3 BLAS and showed how their implementation outperforms the LINPACK [37] and LAPACK [9] implementations[17] on 3 different workstation platforms. Finally, Dayde and Duff in [35] described a new implementation of the Level 3 BLAS for RISC processors (RISC-BLAS library). The Level 3 BLAS are expressed as a sequence of matrix-matrix multiplications and operation involved triangular blocks. Experimental results show that their implementation compares well with the manufacturer-supplied libraries on four different RISC processors. However, all these works optimized the codes by hand and do not try to do it automatically.

17.LAPACK, "Linear Algebra Package", is a project originated by Jack Dongarra. This project put together a new set of linear algebra functions, supposed to supplant both the LINPACK and EISPACK packages. To achieve maximum efficiency across all types of hardware, the LAPACK routines are based on the BLAS3 routines.

5.7 SUMMARY

In the first part of this chapter we have discussed the effects of tiling for different memory level and presented quantitative data showing the benefits of tiling only for the register level, tiling only for the cache level and tiling for both levels simultaneously. Several conclusions can be extracted from this part:

- Tiling for the register level provides two main benefits: (1) it exploits temporal data reuse that translates into a significant reduction of the number of load/store instructions issued and (2) it exploits a program's ILP.
- Moreover, tiling for the register level also diminishes the overall cache misses because reducing the number of load/store instructions reduces data memory traffic.
- However, tiling for the register level can heavily increase the overall TLB misses if spatial locality is not properly exploited. Nevertheless, despite this increase in TLB misses high performance can be achieved by tiling only for the register level.
- At the cache level, tiling is effective for reducing the capacity cache miss rate and thus, average memory access time is reduced. When tiling includes the cache level, a stable performance level is achieved, mostly independent of the problem size.
- However, tiling only for the cache level achieves worse performance than tiling only for the register level. It is much more important to reduce load/store instructions and to increment ILP by tiling for the register level than to reduce the overall cache misses by exploiting the cache level.
- Tiling for both the register and cache levels yields better performance that tiling for any one of them alone, since it achieves 4 goals: (1) ILP is exploited, (2) the number of load/stores instructions is significantly reduced, (3) capacity cache misses are moderated and, finally, (4) TLB misses are also reduced.

In the second part of the chapter, we have compared automatic-optimized codes against hand-optimized codes. The automatic-optimized codes were generated using the compiler algorithms described in Chapter 3 and Chapter 4 for applying multilevel tiling to non-rectangular loop nests and, as hand-optimized codes, we have used two different libraries: the BLAS3 library provided by the manufacturers and the RISC-BLAS library proposed by [34][35]. Results show how compiler technology can make it possible for complex numerical codes to achieve as high performance as hand-optimized codes on modern microprocessors.

6

CONCLUSIONS AND FUTURE WORK

Summary

This chapter summarizes the main contributions of this thesis. Future lines of work opened up by the work presented here are also discussed.

6.1 CONCLUSIONS

We started this thesis asking ourselves why current commercial compilers still perform poorly when dealing with complex scientific applications. In Chapter 1, we showed that, despite all the effort put in current compilers to achieve high performance in numerical codes, hand-optimized codes still outperform them. Moreover, the performance difference between hand-optimized codes and automatic-optimized codes is more noteworthy in complex numerical codes.

We believe that restructuring a code to achieve high performance should be the job of the compiler. Compilers, not programmers, should handle the machine-specific details required to attain high performance on each particular architecture. The main motivation of this thesis were to develop new compilation techniques that address the lack of performance of complex numerical codes consisting of non-rectangular loop nests (loop nests defining non-rectangular iteration spaces).

To fully realize all recent architectural advances and, therefore, achieve high performance on modern superscalar processors, compilers need to find ILP to utilize machine resources effectively, and they also need to transform the program to achieve a high degree of data locality to maximize the effectiveness of the memory system. To this end, several compiler strategies, such as loop tiling, inner unrolling, software pipelining, loop permutation, scalar replacement, unroll-and-jam, etc., have been developed and combined together. From all these compiler strategies, *loop tiling* is the transformation that individually achieves more performance, because it is capable to achieve three main goals:

- improves a program's ILP, when it is applied at the register level,
- exploits data reuse in *several* dimensions of the iteration space and
- enhances data locality at *several* memory levels simultaneously (*multilevel tiling*).

The other loop transformations, if taken individually, realize some of the goals that loop tiling achieves, but not as many. In this thesis we have focused on the loop tiling transformation and our goal was the improvement of the loop tiling transformation when dealing with complex numerical codes.

We observed that the main reason why current commercial compilers perform poorly when dealing with this type of codes is that they do not apply tiling to the register level. Instead, to enhance locality at this level and to improve ILP, they use or combine other transformations, that do not exploit the register level as well as loop tiling.

Tiling for the register level has not generally been considered because the transformed loop nest is not easy to rewrite. At the register level, after dividing the original iteration space into tiles, it is necessary to rewrite the loop body by fully unrolling the loops that traverse the iterations inside the register tiles, because registers are only addressable using absolute addresses. In non-rectangular loop nests, the action of fully unrolling the loops is far from being trivial due to the irregular nature of the iteration space.

Our first main contribution in this thesis has been a general compiler algorithm to perform tiling at the register level that handles arbitrary iteration space shapes and not only simple rectangular shapes.

Our method includes a very simple heuristic to make the tile decisions for the register level. At first sight, register tiling should be performed so that whichever loop carries the most temporal reuse is not tiled. This way, register reuse is maximized and the number of load/store instructions executed is minimized. However, we have shown that, for complex loop nests, if we only consider reuse directions and do not take into account the iteration space shape, the tiled loop nest can suffer performance degradation.

Our second contribution has been a proposal of a very simple heuristic to determine the tiling parameters for the register level, that considers not only temporal reuse, but also the iteration space shape. Moreover, the heuristic is simple enough to be suitable for automatic implementation by compilers.

We evaluated the performance obtained by our method, using as benchmarks typical linear algebra algorithms having non-rectangular iteration spaces. We compared our proposal against commercial compilers and preprocessors able to perform optimizing code transformations such as inner unrolling, unroll-and-jam and software pipelining, on two different superscalar microprocessors (an ALPHA 21164 and a MIPS R10000). In general, our method outperforms both the native compilers and the KAP preprocessor, showing speedups up to 2.9.

However, to be able to achieve similar performance to hand-optimized codes, it is not enough by tiling only for the register level. With today's architectures having complex memory hierarchies and multiple processors, it is quite common that the compiler has to perform tiling at four or more levels (parallelism, L2-cache, L1-cache and registers) in order to achieve high performance. Therefore, in today's architectures it is crucial to have an efficient algorithm that can perform multilevel tiling at multiple levels of the memory hierarchy. Moreover, as we have shown in Chapter 5, multilevel tiling should always include the register level, as this is the memory hierarchy level that yields most performance when properly tiled.

When multilevel tiling includes the register level, it is critical to compute *exact* loop bounds and to avoid the generation of *redundant* bounds. The reason is that the complexity and the amount of code generated by our register tiling technique both depend polynomially on the number of bounds of the loops that have to be fully unrolled (the innermost loops after multilevel tiling). Moreover, computing exact and non-redundant bounds would be also beneficial to avoid increasing a program's execution time.

However, to date, the drawback of generating exact loop bounds and eliminating redundant bounds was that all techniques known were extremely expensive in terms of compilation time and, thus, difficult to integrate in a production compiler.

Our third contribution in this thesis has been a new implementation of multilevel tiling (SMT) that computes exact loop bounds at a much lower complexity than traditional techniques. In fact, we evaluated analytically the complexity of our implementation and showed that it is proportional to the complexity of performing a loop permutation in the original loop nest (before tiling), while conventional techniques have much larger complexities. Moreover, our implementation generates less redundant bounds in the multilevel tiled code and allows removing the remaining redundant bounds in the innermost loops at a lower cost.

We evaluated experimentally our implementation and compared it against traditional techniques in terms of complexity, redundant bounds generated and cost of eliminating the remainder redundant bounds. We have shown that SMT is between 1.5 and 2.8 times faster than conventional implementations for simple non-rectangular loop nests, but it can be even 2300 times faster for more complex loop nests that are commonly found in linear algebra programs using banded matrices. Moreover, experimental results also show that SMT generates less redundant bounds than conventional implementations and the task of eliminating redundant bounds in a multilevel tiled code generated with SMT is between 2.2 and 11 times faster than in a code generated with conventional techniques. Overall, the efficiency of SMT makes it possible to integrate multilevel tiling including the register level in a production compiler without having to worry about compilation time.

The last part of this thesis was dedicated to studying the performance of multilevel tiling. We discussed the effects of tiling for different memory levels and presented quantitative data comparing the benefits of tiling only for the register level, tiling only for the cache level and tiling for both levels simultaneously. The main conclusions of this discussion were that:

- tiling for the register level provides two main benefits: it significantly reduces the number of load/store instructions issued and exploits a program's ILP. However, we also observed that it can have the drawback of heavily increasing the overall TLB misses,
- tiling for the cache level is effective for reducing the capacity cache miss rate,

- tiling for both the register and cache levels yields better performance that tiling for any one of them alone, since it achieves 4 goals: ILP is exploited, the number of load/stores instructions is significantly reduced, capacity cache misses are moderated and, finally, TLB misses are also moderated and
- when tiling for multiple memory levels, the register level is the most important one, more so than the cache level, although for loop nests with very large working sets, tiling for various cache levels is also important. In general, when performing multilevel tiling the compiler should always include the register level in order to achieve high performance.

Finally, we compared automatically-optimized codes against hand-optimized codes, on two different architectures (ALPHA 21164 and MIPS R10000). The automatically-optimized codes were generated using our own tool that implements the methods proposed for applying multilevel tiling to complex loop nests. For the hand-optimized codes, we used two different libraries: the vendor-supplied BLAS3 library and the RISC-BLAS library. Results showed that our automatically generated codes almost always match the performance of hand-optimized codes. Indeed, for small problem sizes, we almost always outperformed the RISC-BLAS and the vendor-supplied libraries and, for large problem sizes we were always very close to the vendor hand-optimized BLAS3 library. The general conclusion is that compiler technology can make it possible for complex numerical codes to achieve the same performance as hand-optimized codes on modern microprocessors.

6.2 FUTURE WORK

Our work has focused on the high-level (source-to-source) loop tiling transformation to achieve high performance on modern microprocessors. Extensions to our work are the following:

Software Prefetching

One extension to our work is studying the effects of combining loop tiling with software prefetching. Some architectures today include prefetch instructions which enable a program to fetch data into the cache before they are used. Would it be beneficial to combine loop tiling and software prefetching in this type of architectures? The appeal of combining both types of optimizations rests on the fact that tiling *avoids* latency while software prefetching *hides* the leftover latency. Therefore, tiling and software prefetching can be considered complementary optimizations and should be helpful to eliminate the stall time completely.

Data Transformations

Another extension to our work is investigating the interaction of data transformations with loop tiling. It is known that applying locality-enhancing transformations, such as data layout optimizations, before tiling is preferable as it makes performance of the tiling less sensitive to the tile sizes. Data transformations have the advantage that they are almost always legal. However, a drawback is that their effect is global in the sense that decisions regarding the memory layout of an array influence the locality characteristics of every part of the program that accessed the said array. The impact of integrating data transformations on tiling merits further investigation.

Integrating Register Tiling in a Compiler

The algorithm proposed in this thesis to perform register tiling is a high-level (source to source) transformation that helps compilers generate efficient machine code. Working at the source level prevents us from controlling many of the low level transformations typically performed by the compiler's back-end (instruction scheduling, register allocation, etc.), making it difficult (if not impossible) to find the optimal tiling parameters. By integrating our technique inside a production compiler we could be more precise in the selection of the tiling parameters. Moreover, it would also allow us to combine register tiling with software pipelining. Register tiling increases register pressure and prevents compilers to perform software pipelining. An interesting question to answer is how register tiling and software pipelining can be combined together to achieve all the goals provided by each transformation individually.

A

FOURIER-MOTZKIN ALGORITHM

A.1 INTRODUCTION

The Fourier-Motzkin algorithm [16][55][77] is an algorithm that iterates *n-1* times (for a *n*-deep loop nest) and bounds each of the loop iteration variables from innermost to outermost. In each iteration, two different steps are performed:

- In the first step, the bounds of the yet-to-be-processed loops are examined. All bounds that are affine functions of the loop iteration variable being solved become simple bounds of this loop.
- In the second step, each of the lower simple bounds of the iteration variable solved in the first step is compared with each of the upper simple bounds. These comparisons generate inequalities that might become new simple bounds of the yet-to-be-processed loops.

A.2 IMPLEMENTATION

Let $A \cdot x \leq \beta$ be a system of inequalities that represents the bounds of a *n*-deep loop nest written in matrix form. The Fourier-Motzkin algorithm can be implemented as follows:

1 - First step:

(a) Transform the system of inequalities in such a way that all the elements in the column of A associated with the first iteration variable to be bounded have value +1, -1 or 0. To that end, each row of the matrix inequality is divided by the required value. At the end of this step, the system of inequalities can be decomposed into three systems:

$$A_+ \cdot x \leq \beta_+ \qquad A_- \cdot x \leq \beta_- \qquad A_0 \cdot x \leq \beta_0$$

where matrices A_+, A_- and A_0 have 1, -1 and 0, respectively, in the columns associated with the iteration variable to be bounded.

(b) Let x_j be the iteration variable to be bounded. Then, the first two systems of inequalities can be rewritten as follows:

$$A_+ \cdot x \leq \beta_+ \Rightarrow x_j \leq \beta_+ - (\tilde{A}_+ \cdot \tilde{x})$$
$$A_- \cdot x \leq \beta_- \Rightarrow (\tilde{A}_- \cdot \tilde{x}) - \beta_- \leq x_j$$

where $\tilde{A}_+$ is matrix A_+ without the column associated with x_j and $\tilde{x}$ is vector x without component x_j. Note that this transformation is possible since the elements of A_+ which multiply x_j are 1. Analogously, $\tilde{A}_-$ and $\tilde{A}_0$ are defined.

(c) The bounds for x_j are:

$$max\left((\tilde{A}_- \cdot \tilde{x}) - \beta_- \right) \leq x_j \leq min\left(\beta_+ - (\tilde{A}_+ \cdot \tilde{x}) \right) \quad (1)$$

The bounds expressed in this way may take non-integer values and only integer values are allowed for loop iteration variables. Therefore, we take the ceiling function for the lower bound and the floor function for the upper bound.

2 - Second step:

(d) From (1) we build the following matrix inequality:

$$(\tilde{A}_- \cdot \tilde{x}) - \beta_- \leq \beta_+ - (\tilde{A}_+ \cdot \tilde{x}) \quad (2)$$

(e) We take now each row on the left hand of matrix inequality (1) and combine it with every row on the right hand of this matrix inequality. In this way we obtain a set of new inequalities which, when put together with the system $\tilde{A}_0 \cdot \tilde{x} \leq \beta_0$ and removing the redundant inequalities, form the new system:

$$\tilde{A} \cdot \tilde{x} \leq \tilde{\beta}$$

Steps (a) to (e) are repeated to obtain the bounds for the next iteration variables, until vector $\tilde{x}$ has a single component.

B

PROOF

Complexity and Code Expansion (Chapter 3 - Section 3.2.4)

B.1 INTRODUCTION

After dealing with the restrictions that split UCLs, we obtain a set of loop nests (or partitions). In only one of these partitions the bound components of the inner UCL do not depend on the iteration variable of the outer UCL and therefore, this is the only partition where the UCLs can be fully unrolled. Let A and B be the number of (upper and lower) bound components of the outermost and innermost UCLs in this particular partition, respectively. Neither A nor B contain the bound components that we want to be the *effective* ones[1].

Every time index set splitting is applied to a loop, it decomposes it into two new loops, generating two partitions. In the two generated partitions, a new bound component appears in the loop being split and there is a bound component of an UCL that is redundant and can be removed. The loop being split cannot be an UCL, because we have already dealt with the restrictions that split UCLs. Thus, the number of bound components in the UCLs will never increase. Then, to achieve the partition where both UCLs can be fully unrolled, ISS must be applied A+B times, that is, as many times as bound components are in both UCLs, ignoring the bound components that we want to be the *effective* ones.

Moreover, the number of loop nests obtained after applying ISS A+B times (to achieve the partition where both UCLs can be fully unrolled) is A+B+1 and in only one of these loop nests both UCLs can be fully unrolled. In the other A+B loop nests, we will be able to unroll one (or none) of the two innermost loops by applying again ISS (repeatedly) to each of these loop nests. At each of the loop nests ISS is performed different number of times; it depends on the number of bound components that the UCL has.

As an example, Figure B.1 shows how the loop nests are generated when ISS is applied repeatedly in order to fully unroll the UCLs in all partitions. A_1, A_2 and B_1, B_2 are the number of lower and upper bound components of the outermost and innermost UCL, respectively, and JJ, $JJ+B_{JJ}-1$ and II, $II+B_{II}-1$ are the lower and upper bound components of the outermost and innermost UCLs, respectively, that we want to be the effective ones. We have assumed $A_1=A_2=B_1=B_2=1$, thus $A=A_1+A_2=2$ and $B=B_1+B_2=2$. To represent the ISS process we use a binary tree. Each node represents a loop nest and its two sons are the partitions obtained after applying ISS with a particular restriction. At each node, we only show the loop bounds of the UCLs. The first row shows the loop bound components of the outermost UCL and the second row shows the loop bound components of the innermost UCL. The lower and upper bound components are in the first and second column, respectively. The leafs of the tree are all the partitions we get at the final code and the loops marked in bold are the loops (UCLs) that can be fully unrolled.

1.Recall from Section 3.2.4 that, for simplicity, the expressions are developed for two UCLs in the loop nest, but they can be easily extended for any number of UCLs.

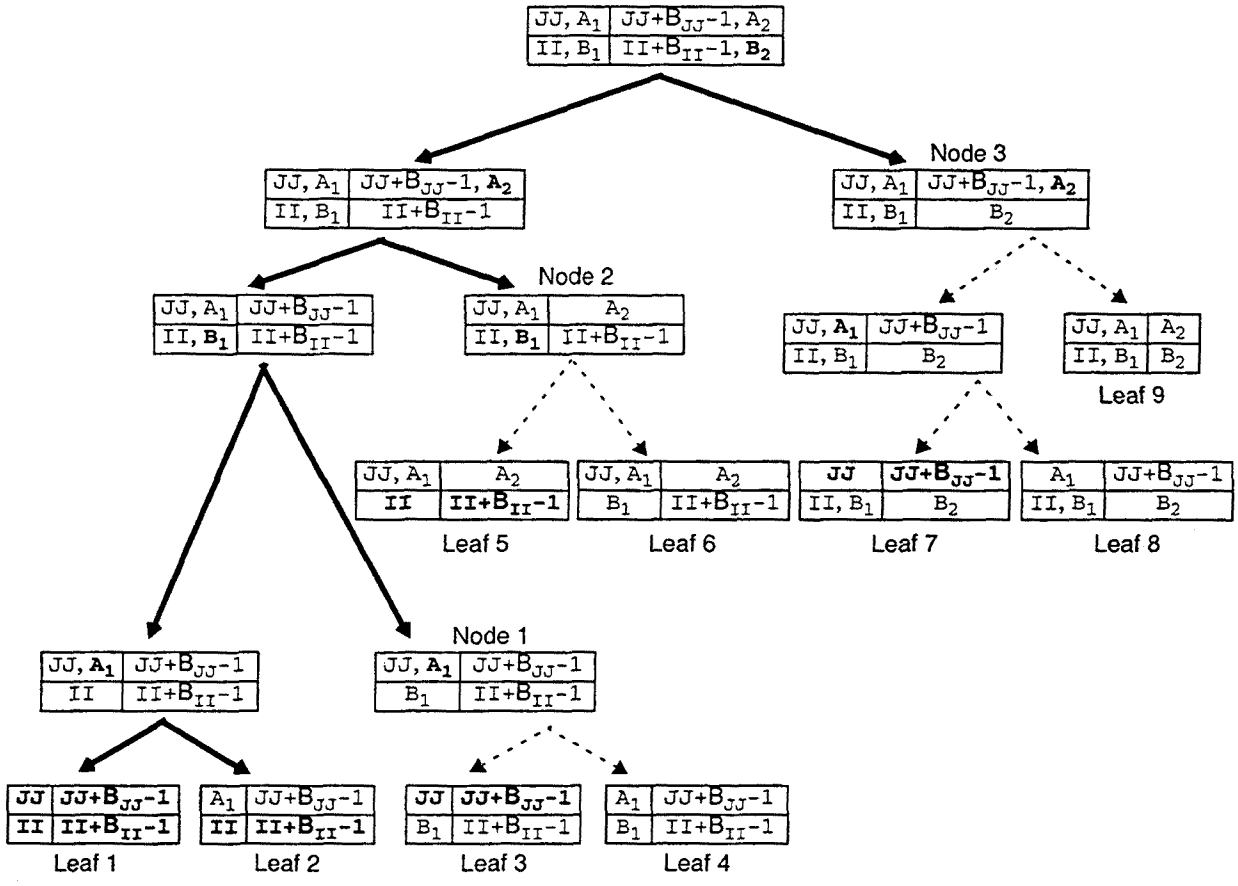

Figure B.1: Example of how the loop nests are generated when Index Set Splitting is applied repeatedly in order to fully unroll the UCLs in all partitions.

The order in which the loop bound components of the UCLs are eliminated, depends on the order in which the restrictions are applied. In Section 3.2.3 of Chapter 3 we show that the order in which we deal with each restriction, that is, the order in which we split the loops is very important to avoid processing a loop more than once and to reduce code expansion. We propose to deal with the restrictions that split loops that are not UCLs from outermost to innermost. In Fig. B.1 we have assumed that, using this processing order of the restrictions, the loop bound components are eliminated in the following order: first $\{B_2\}$, second $\{A_2\}$, third $\{B_1\}$ and finally $\{A_1\}$. At each node we have marked in bold the loop bound component that is going to be eliminated from one of the two partitions after applying ISS.

To achieve the partition where both UCLs can be fully unrolled (Leaf 1), ISS has been applied `A+B=4` times (dark lines) and `A+B+1=5` partitions (or loop nests) have been obtained (Leaf 1, Leaf 2, Node 1, Node 2 and Node 3). In `A+B=4` loop nests (Leaf 2, Node 1, Node 2 and Node 3) only one of the UCLs can be fully unrolled. From now on, we will refer to these loop nests as *partially unrolled loop*

nests. In Node 1 and 3 the outermost loop is still an UCL and in Node 2 and Leaf 2 the innermost loop is still an UCL. Thus, we can apply ISS again to these loop nests (dashed lines in Fig. B.1) to be able to unroll the corresponding UCLs.

On each of the *partially unrolled loop nests*, ISS is applied a different number of times. In Node 2, for example, ISS is applied only once, while in Node 3 it is applied twice. In this appendix, we will demonstrate that the sum of the number of times ISS is applied on all the partially unrolled loop nests is always A*B.

B.2 PROOF

Definition 1

The loop nest obtained after dealing with the restrictions that split UCLs can be written, in the general case, into the following form:

```
do JJ = ...
  do II = ...
    ...
UCLs do J = max (JJ, {A1}), min(JJ+BJJ-1, {A2})        ┐
       do I = max({B1}, II), min(II+BII-1, {B2})       │ partition where the bound
         loop body                                     │ components of I are not
       enddo                                           │ affine functions of J
     enddo                                             ┘
     do J = ...                                        ┐
       do I = ...                                      │ partitions where at least
         loop body                                     │ one bound component of I
       enddo                                           │ is an affine function of J
     enddo                                             ┘
  ...
enddo
```

where loops I and J, in the first partition, are the UCLs and A_1, A_2 and B_1, B_2 are the number of (lower and upper) bound components of the outermost and innermost UCLs, respectively. Neither A_1, A_2 nor B_1, B_2 contain the bound components that we want to be the *effective* ones. Moreover, since we have already dealt with the restrictions that split UCLs, the B_1+B_2 bound components of I cannot be affine functions of J.

Definition 2

Let A and B be the total number of bound components of the outermost and innermost UCLs, respectively, ignoring the bound components that we want to be the *effective* ones. Thus, $A=A_1+A_2$ and $B=B_1+B_2$.

Theorem

Let's define $f(A, B)$ as the total number of times that ISS is applied to be able to unroll the UCL in all *partially unrolled loop nests*, then $f(A, B) = A*B$.

Proof (by induction)

Clearly, $f(0, B) = f(A, 0) = 0$, because in all partially unrolled loop nests, the UCL can be directly fully unrolled. Also clearly, it can be seen that, if A=B=1, then:

$$f(1,1) = \begin{cases} 1 + f(0,1) = 1 & \text{if the bound component of loop J is eliminated first} \\ 1 + f(1,0) = 1 & \text{if the bound component of loop I is eliminated first} \end{cases}$$

Thus, the basis is trivial, since $f(0, B) = 0*B = 0$, $f(A, 0) = A*0 = 0$ and $f(1, 1) = 1*1 = 1$.

Let's assume that the proof holds for $f(A, B-1)$ and $f(A-1, B)$, that is, $f(A, B-1) = A*(B-1)$ and $f(A-1, B) = (A-1)*B$.

Suppose that the first time ISS is applied to the original code, a bound component of loop J is eliminated. After applying ISS, two partitions are generated. In one of them, both UCLs (I and J) can still be fully unrolled and loop J has one fewer bound component. In the other partition (the partially unrolled loop nest), only loop I can still be fully unrolled and it still has the same bound components as before applying ISS, that is, B bound components. Therefore, ISS has to be applied B times to the partially unrolled loop nest to achieve the partition where loop I can be fully unrolled.

Let's now suppose the other case, that is, the first time ISS is applied to the original code, a bound component of loop I is eliminated. As before, after applying ISS, two partitions are generated. In one of them, both UCLs (I and J) can still be fully unrolled and now loop I has one bound component less. In the other partition (the partially unrolled loop nest), only loop J can still be fully unrolled and it still has the same bound components as before applying ISS, that is, A bound components. Therefore, ISS has to be applied A times to the partially unrolled loop nest to achieve the partition where loop J can be fully unrolled. Therefore,

$$f(A,B) = \begin{cases} B + f(A-1,B) & \text{if the first eliminated bound component corresponds to loop J} \\ A + f(A,B-1) & \text{if the first eliminated bound component corresponds to loop I} \end{cases}$$

Replacing $f(A, B-1)$ and $f(A-1, B)$ by their values, we obtain:

$$f(A,B) = \begin{cases} B + f(A-1,B) = B + (A-1)\cdot B = A\cdot B \\ A + f(A,B-1) = A + (B-1)\cdot A = A\cdot B \end{cases}$$

■

B.3 BACK TO THE NOTATION OF CHAPTER 3

Turning to the notation used in Section 3.2.4 of Chapter 3, after dealing with the `M` restrictions that split UCLs, we obtain one partition where the bound components of the inner UCL do not depend on the iteration variable of the outer UCL. In this partition, the outer UCL has, in the worst case, `M+R` bound components and the inner UCL has `S` bound components. Therefore, the sum of the number of times ISS is performed on each of the loop nests where one UCL can still be fully unrolled (partially unrolled loop nests) is, in the worst case, `(M+R)*S`.

C

BENCHMARK PROGRAMS

MMtri: Triangular Matrix Product

```
do K = 1, N
  do J = K, N
    do I = K, N
      C(I,J) = C(I,J) + A(I,K) * B(K,J)
    enddo
  enddo
enddo
```

LU: LU Decomposition without Pivoting

```
do K = 1, N-1
  do I = K+1, N
    A(I,K) = A(I,K) / A(K,K)
  enddo
  do J = K+1, N
    do I = K+1, N
      A(I,J) = A(I,J) - A(I,K) * A(K,J)
    enddo
  enddo
enddo
```

CHOL: Cholesky Factorization

```
do K = 1, N
  A(K,K) =  SQRT(A(K,K))
  do I = K+1, N
    A(I,K) = A(I,K) / A(K,K)
  enddo
  do J = K+1, N
    do I = J, N
      A(I,J) = A(I,J) - A(I,K) * A(J,K)
    enddo
  enddo
enddo
```

QR: Givens QR-decomposition (skewed code)

```
do J = 1, N-1
  do I = J-N, -1
    if (A(J-I,J) .eq. 0) then
      C(I,J) = 1.0
      S(I,J) = 0.0
    else if (ABS(A(J-I-1,J)) .le. ABS(A(J-I,J))) then
      T = -A(J-I-1,J) / A(J-I,J)
      S(I,J) = 1.0 / DSQRT(1+T*T)
      C(I,J) = S(I,J)*T
    else
      T = -A(J-I,J) / A(J-I-1,J)
      C(I,J) = 1.0 / DSQRT(1+T*T)
      S(I,J) = C(I,J)*T
    endif
    T1 = A(J-I-1,J)
    T2 = A(J-I,J)
    A(J-I-1,J) = C(I,J) * T1 - S(I,J) * T2
    A(J-I,J) = S(I,J) * T1 + C(I,J) * T2
  enddo
  do K = J+1, N
    do I = J-N, -1
      T1 = A(J-I-1,K)
      T2 = A(J-I,K)
      A(J-I-1,K) = C(I,J) * T1 - S(I,J) * T2
      A(J-I,K) = S(I,J) * T1 + C(I,J) * T2
    enddo
  enddo
enddo
```

SSYR2K: Symmetric Rank 2k Update

```
do K = 1, N
  do J = 1, N
    do I = 1, J
      C(I,J) = C(I,J) + B(J,K) * A(I,K) + A(J,K) * B(I,K)
    enddo
  enddo
enddo
```

SSYRK: Symmetric Rank k Update

```
do K = 1, N
  do J = 1, N
    do I = J, N
      C(I,J) = C(I,J) + A(J,K) * A(I,K)
    enddo
  enddo
enddo
```

SSYMM: Symmetric Matrix-Matrix Operation

```
do J = 1, N
  do I = 1, N
    do K = 1, I-1
      C(K,J) = C(K,J) + A(K,I) * B(I,J)
      C(I,J) = C(I,J) + A(K,I) * B(K,J)
    enddo
      C(I,J) = C(I,J) + A(I,I) * B(I,J)
  enddo
enddo
```

STRMM: Product of Triangular and Square Matrix

```
do K = 2, N
  do J = 1, N
    do I = 1, K-1
      B(I,J) = B(I,J) + A(I,K) * B(K,J)
    enddo
  enddo
enddo
```

STRSM: Solve a Matrix Equation

```
do K = 1, N
  do J = 1, N
    B(K,J) = B(K,J) / A(K,K)
    do I = K+1, N
      B(I,J) = B(I,J) - A(I,K) * B(K,J)
    enddo
  enddo
enddo
```

D

PROOF

Theorem 2
(Chapter 4 - Section 4.4.1)

Theorem

In SMT, the second step of the FM algorithm does not need to be performed when the loop being solved is a TI-loop.

Proof [1]

We will demonstrate that, when computing the bounds of a TI-loop, all new simple bounds generated by the second step of the FM algorithm are redundant. Therefore, the second step of the FM algorithm does not need to be performed when the loop being solved is a TI-loop.

Let the following loop nest be the original code:

```
do I = L_I, U_I
  do K = α1 · I + θ1 , U_K
    do J = L_J, α2 · I + θ2
enddo
```

where $\alpha_1, \alpha_2, \theta_1, \theta_2$ are integer constants or program parameters (variables unchanged within the loops). To simplify the proof, we assume that L_I, L_J and U_I, U_K are max and min compositions of integers constants or program parameters, respectively. We note that if L_J and U_K were affine functions that depend of outer loop index variables (I or K), the demonstration would be done in a similar way.

Let's also assume that $\alpha_1, \alpha_2 > 0$ and $\alpha_1 \geq \alpha_2$ (for other values of α_1 and α_2 the demonstration is done in a similar way). Let's also assume that all loops, I, J and K, have to be strip-mined only once with tile sizes $B_{II} > 0$, $B_{JJ} > 0$ and $B_{KK} > 0$, respectively, and the desired loop order in the multilevel tiled code (BTIS) is KK-JJ-II-K-J-I, from outermost to innermost (for more levels of tiling and different loop orders, the demonstration is the same).

From Theorem 1 of Chapter 4, we have that the bounds of the Minimum Convex Space of the BTIS are extracted from the matrix inequality $A^{NCBIS} \cdot \left(T^P\right)^{-1} \cdot \mathtt{I}^{BTIS} \leq \beta^{NCBIS}$ using the Fourier-Motzkin algorithm, where $A^{NCBIS} \cdot \mathtt{I}^{NCBIS} \leq \beta^{NCBIS}$ are the bounds of the MCS of the NCBIS (the loop nest after strip-mining all loops at all levels) and T^P defines the loop permutation transformation. From Lemma 4 and Corollary 3 (Section 4.6 of Chapter 4) we have that, if strip-mining is applied first to all loops at all levels and then the loop permutation is performed, *special redundant bounds* will be generated in the tiled code.

To avoid the generation of these redundant bounds, and without a loss of generality, we will compute the bounds of the BTIS in the following way[2]: first, we apply strip-mining only to loop I and perform one iteration of the FM algorithm to obtain the bounds of I in the BTIS; second, we apply strip-mining only to loop J and perform one iteration of FM to obtain the bounds of J; third, we apply

1.For a clear understanding of some assumptions of this proof, we recommend reading first Section 4.6 of Chapter 4.
2.Our efficient implementation of SMT (Section 4.4.4 of Chapter 4) will also compute the bounds in this way.

strip-mining to loop K and perform one iteration of FM, obtaining the bounds of K in BTIS; and fourth, we perform the required iterations of FM to compute the bounds of TI-loops II, JJ and KK.

In the demonstration, we will present the bounds of the loops at intermediate steps of the tiling transformation. More precisely, we will present the MCS defined by the bounds of the loops and we will present this information in the form of a loop nest (we use braces on the loop bounds to indicate that they are representing the MCS). Moreover, once the bounds of a particular loop have been computed, the loop is moved to its position in the final tiled code. Thus, the loops whose bounds have already been computed are always in the innermost positions of the nest while the not-yet-processed loops are in the outermost positions.

After the bounds of the EL-loops have been computed as mentioned above, the following bounds are obtained:

$$\text{do II} = \{ L_I\text{-}B_{II}\text{+}1,\ \min(U_I,\ \left\lfloor \frac{U_K-\theta_1}{\alpha_1} \right\rfloor) \}$$

$$\text{do KK} = \{ \max(\alpha_1 \cdot L_I + \theta_1,\ \alpha_1 \cdot II + \theta_1)\text{-}B_{KK}\text{+}1,\ U_K \}$$

$$\text{do JJ} = \{ L_J\text{-}B_{JJ}\text{+}1,\ \min(\alpha_2 \cdot U_I + \theta_2,\ \alpha_2 \cdot (II + B_{II} - 1) + \theta_2,\ \left\lfloor \frac{\alpha_2}{\alpha_1} \cdot (U_K - \theta_1) \right\rfloor + \theta_2,\ \left\lfloor \frac{\alpha_2}{\alpha_1} \cdot (KK + B_{KK} - 1 - \theta_1) \right\rfloor + \theta_2) \}$$

$$\left.\begin{array}{l} \text{do K} = \{ \max(KK,\ \alpha_1 \cdot L_I + \theta_1,\ \alpha_1 \cdot II + \theta_1,\ \left\lceil \frac{\alpha_1}{\alpha_2} \cdot (JJ - \theta_2) \right\rceil + \theta_1),\ \min(U_K,\ KK\text{+}B_{KK}\text{-}1) \} \\ \text{do J} = \{ \max(JJ,\ L_J),\ \min(JJ\text{+}B_{JJ}\text{-}1,\ \alpha_2 \cdot U_I + \theta_2,\ \alpha_2 \cdot (II + B_{II} - 1) + \theta_2,\ \left\lfloor \frac{\alpha_2}{\alpha_1} \cdot (K - \theta_1) \right\rfloor + \theta_2) \} \\ \text{do I} = \{ \max(L_I,\ II,\ \left\lceil \frac{J-\theta_2}{\alpha_2} \right\rceil),\ \min(U_I,\ II\text{+}B_{II}\text{-}1,\ \left\lfloor \frac{K-\theta_1}{\alpha_1} \right\rfloor) \} \end{array}\right] \text{EL-loops}$$

At this point, we want to compute the bounds of the TI-loop II in the BTIS. The first step of the FM examines the simple bounds of the not-yet-processed loops and all simple bounds that are affine functions of the loop iteration variable being solved become simple bounds of this loop. Therefore, the simple bounds of II in the BTIS are its current simple bounds plus the simple bounds of loops JJ and KK that are affine functions of II. After performing the first step of FM the following bounds are obtained:

$$\text{do KK} = \{ \alpha_1 \cdot L_I + \theta_1 - B_{KK}\text{+}1,\ U_K \}$$

$$\text{do JJ} = \{ L_J\text{-}B_{JJ}\text{+}1,\ \min(\alpha_2 \cdot U_I + \theta_2,\ \left\lfloor \frac{\alpha_2}{\alpha_1} \cdot (U_K - \theta_1) \right\rfloor + \theta_2,\ \left\lfloor \frac{\alpha_2}{\alpha_1} \cdot (KK + B_{KK} - 1 - \theta_1) \right\rfloor + \theta_2) \}$$

$$\text{do II} = \{ \max(L_I,\ \left\lceil \frac{JJ-\theta_2}{\alpha_2} \right\rceil)\text{-}B_{II}\text{+}1,\ \min(U_I,\ \left\lfloor \frac{U_K-\theta_1}{\alpha_1} \right\rfloor,\ \left\lfloor \frac{KK + B_{KK} - 1 - \theta_1}{\alpha_1} \right\rfloor) \}$$

do K = . . ., . . .

do J = . . ., . . .

do I = . . ., . . .

The second step of the FM algorithm consists in comparing each of the lower simple bounds of the iteration variable solved in the first step with each of the upper simple bounds. These comparisons generate inequalities that might become new simple bounds of the yet-to-be-processed loops. By comparing the lower simple bounds of II with its upper simple bounds, we obtain the following inequalities where loops JJ and/or KK are involved:

1- $\left\lceil \frac{JJ-\theta_2}{\alpha_2} \right\rceil - B_{II} + 1 \le U_I$

2- $\left\lceil \frac{JJ-\theta_2}{\alpha_2} \right\rceil - B_{II} + 1 \le \left\lfloor \frac{U_K-\theta_1}{\alpha_1} \right\rfloor$

3- $\left\lceil \frac{JJ-\theta_2}{\alpha_2} \right\rceil - B_{II} + 1 \le \left\lfloor \frac{KK+B_{KK}-1-\theta_1}{\alpha_1} \right\rfloor$

4- $L_I - B_{II} + 1 \le \left\lfloor \frac{KK+B_{KK}-1-\theta_1}{\alpha_1} \right\rfloor$

These inequalities generate new simple bounds on loops JJ or KK that are trivially redundant[3]:

1- $JJ \le \alpha_2 \cdot (U_I + B_{II} - 1) + \theta_2$: This bound is redundant because loop JJ already has the bound $JJ \le \alpha_2 \cdot U_I + \theta_2$ and B_{II} is positive.
Therefore, $\min(\alpha_2 \cdot U_I + \theta_2,\ \alpha_2 \cdot (U_I + B_{II} - 1) + \theta_2) = \alpha_2 \cdot U_I + \theta_2$

2- $JJ \le \left\lfloor \frac{\alpha_2}{\alpha_1} \cdot (U_K - \theta_1) \right\rfloor + \theta_2 + \alpha_2 \cdot (B_{II} - 1)$: This bound is redundant because loop JJ already has the bound $JJ \le \left\lfloor \frac{\alpha_2}{\alpha_1} \cdot (U_K - \theta_1) \right\rfloor + \theta_2$ and α_2 and B_{II} are positive.
Therefore, $\min(\left\lfloor \frac{\alpha_2}{\alpha_1} \cdot (U_K - \theta_1) \right\rfloor + \theta_2,\ \left\lfloor \frac{\alpha_2}{\alpha_1} \cdot (U_K - \theta_1) \right\rfloor + \theta_2 + \alpha_2 \cdot (B_{II} - 1)) = \left\lfloor \frac{\alpha_2}{\alpha_1} \cdot (U_K - \theta_1) \right\rfloor + \theta_2$

3- $JJ \le \left\lfloor \frac{\alpha_2}{\alpha_1} \cdot (KK + B_{KK} - 1 - \theta_1) \right\rfloor + \theta_2 + \alpha_2 \cdot (B_{II} - 1)$: This bound is redundant because loop JJ already has the bound $JJ \le \left\lfloor \frac{\alpha_2}{\alpha_1} \cdot (KK + B_{KK} - 1 - \theta_1) \right\rfloor + \theta_2$ and α_2 and B_{II} are positive.
Therefore, $\min(\left\lfloor \frac{\alpha_2}{\alpha_1} \cdot (KK + B_{KK} - 1 - \theta_1) \right\rfloor + \theta_2,\ \left\lfloor \frac{\alpha_2}{\alpha_1} \cdot (KK + B_{KK} - 1 - \theta_1) \right\rfloor + \theta_2 + \alpha_2 \cdot (B_{II} - 1)) =$
$= \left\lfloor \frac{\alpha_2}{\alpha_1} \cdot (KK + B_{KK} - 1 - \theta_1) \right\rfloor + \theta_2$

4- $KK \ge \alpha_1 \cdot (L_I - B_{II} + 1) + \theta_1 - B_{KK} + 1$: This bound is redundant because loop KK has the bound $\alpha_1 \cdot L_I + \theta_1 - B_{KK} + 1$ and B_{II} is positive.
Therefore, $\max(\alpha_1 \cdot L_I + \theta_1 - B_{KK} + 1,\ \alpha_1 \cdot (L_I - B_{II} + 1) + \theta_1 - B_{KK} + 1) = \alpha_1 \cdot L_I + \theta_1 - B_{KK} + 1$

Thus, all new simple bounds produced by the second step of FM when computing the bounds of TI-loop II are redundant. ■

3.A simple bound of a particular loop I is a *trivial* redundant simple bound if it can be deduced that it is redundant by only looking at the other simple bounds of loop I.

BIBLIOGRAPHY

[1] W. Abu-Sufah. "Improving the performance of virtual memory computers". Ph.D. Thesis, University of Illinois at Urbana-Champaign, November 1978.

[2] A. Agarwal. "Performance trade-offs in multithreaded processors". IEEE Transactions on Parallel and Distributed Systems, Vol. 3, No. 5, pp. 525-539, September 1992.

[3] A. Agarwal and S. Pudar. "Column-associative caches: a technique for reducing the miss rate of direct-mapped caches". In Proceedings of the 20th International Symposium on Computer Architecture, pp.179-190, 1993.

[4] E. L. Allgower. "Exact inverse of certain band matrices". Numerical Mathematics, Vol. 21, pp. 279-284, 1973.

[5] S. P. Amarasinghe. "Parallelizing compiler techniques based on linear inequalities". Ph.D. Thesis, Stanford University, January 1997.

[6] S. P. Amarasinghe and M. S. Lam. "Communication optimization and code generation for distributed memory machines". In Proceedings of the ACM SIGPLAN' 93 Conference on Programming Language Design and Implementation, pp. 126-138, June 1993.

[7] C. Ancourt and F. Irigoin. "Scanning polyhedra with DO loops". In Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 39-50, 1991.

[8] D. Anderson, F. Sparacio and F. Tomasulo. "The IBM system/360 model 91: machine philosophy and instruction handling". IBM Journal of Research and Development, Vol. 11, pp. 8-24, January 1967.

[9] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Grennbaum, S. Hammarling, A. McKenney, S. Ostrouchov and D. Sorensen. "LAPACK user's guide". 2nd edition, SIAM Press, 1995.

[10] J. Anderson, S. Amarasinghe and M. Lam. "Data and computation transformations for multiprocessors". In Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, July 1995.

[11] D. Bacon, S. Graham and O. Sharp. "Compiler transformations for high-performance computing". Computing Surveys, Vol. 26, No. 4, pp. 345-420, 1994.

[12] J. Baer and T. Chen. "An effective on-chip preloading scheme to reduce data access penalty". In Supercomputing'91, pp. 176-186, 1991.

[13] U. Banerjee. "Unimodular transformation of double loops". In Advances in Languages and Compilers for Parallel Processing, pp. 192-219. The MIT Press, 1991.

[14] U. Banerjee. "Dependence analysis for supercomputing". Norwell, Mass. Kluwer Academic Publishers, 1988.

[15] D. P. Bhandarkar. "Alpha implementations and architecture: complete reference and guide". Digital Press, 1996.

[16] A. Bik, H.A.G. Wijshoff. "Implementation of Fourier-Motzkin elimination". Technical Report 94-42, Department. of Computer Science, Leiden University, 1994.

[17] A. Bik, H.A.G. Wijshoff. "Iteration space partitioning". In Proceedings of the International Conference on High Performance Computing and Networking, pp. 475-484, 1996.

[18] P. Boulet, A. Darte, T. Risset and Y. Robert. "(Pen)-ultimate tiling?". In Integration, the VLSI Journal, Vol. 17, pp. 33-51, 1994.

[19] B. Calder, D. Grunwald and J. Emer. "Predictive sequential associative cache". In Proceedings of the 2nd International Symposium on High-Performance Computer Architecture, pp. 244-253, February 1996.

[20] D. Callahan, S. Carr and K. Kennedy. "Improving register allocation for subscripted variables". International Conference on Programming Language Design and Implementation, pp. 53-65, June 1990.

[21] D. Callahan, J. Cocke and K. Kennedy. "Estimating interlock and improving balance for pipelined architectures". Journal of Parallel and Distributed Computing, pp. 334-358, 1988.

[22] D. Callahan, K. Kennedy and A. Porterfield. "Software prefetching". In Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 40-52, April 1991.

[23] S. Carr. "Memory-hierarchy management". Ph.D. Thesis, Rice University, February 1993.

[24] S. Carr. "Combining optimization for cache and Instruction-Level Parallelism". In Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques, pp. 238-247, October 1996.

[25] S. Carr and K. Kennedy. "Compiler blockability of numerical algorithms". ACM International Conference on Supercomputing, pp. 114-124, 1992.

[26] S. Carr and K. Kennedy. "Improving the ratio of memory operations to floating-point operations in loops". ACM Transactions on Programming Languages and Systems, Vol. 16, No. 6, pp. 1768-1810, November 1994.

[27] S. Carr, K. McKinley and C-W. Tseng. "Compiler optimizations for Improving Data Locality". International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 252-262, August 1994.

[28] L. Carter, J. Ferrante and S. Flynn Hummel. "Hierarchical tiling for improved superscalar performance". In the 9th International Parallel Processing Symposium, April 1995.

[29] D. Chen. "Hierarchical blocking and data flow analysis for numerical linear algebra". ACM International Conference on Supercomputing, pp. 12-19, 1991.

[30] T. F. Chen and J. L. Baer. "Effective hardware-based data prefetching for high performance processors". IEEE Transactions on Computers, Vol. 44, No. 5, pp. 609-623, May 1995.

[31] S. Coleman and K. McKinley. "Tile size selection using cache organization and data layout". In Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation, June 1995.

[32] T. Coleman and C. Van Loan. "Handbook for matrix computation". SIAM, Philadelphia, 1988.

[33] E. Cuthill. "Several strategies for reducing the bandwidth of matrices". In D.J. Rose and R.A. Willoughby, editors, Sparse Matrices and Their Applications. Plenum Press, 1972.

[34] M. Dayde and I. Duff. "A block implementation of level 3 BLAS for RISC processors". Technical Report RT/APO/96/1, ENSEEIHT-IRIT, April 1996.

[35] M. Dayde and I. Duff. "The RISC BLAS: A blocked implementation of level 3 BLAS for RISC processors". Technical Report RT/APO/98, ENSEEIHT-IRIT, January 1998.

[36] Digital Equipment Corporation. "ATOM user manual", 1995

[37] J. Dongarra, J. Bunch, C. Moler and W. Stewart. "LINPACK user's guide". SIAM Press, 1979

[38] J. Dongarra, J.D. Croz, S. Hammarling and R. Hanson. "An extended set of fortran Basic Linear Algebra Subprograms". ACM Transactions on Mathematical Software, Vol. 14, pp. 1-17, 1988.

[39] J. Dongarra, J.D. Croz, S. Hammarling and I. Duff. "A set of Level 3 Basic Linear Algebra Subprograms". ACM Transactions on Mathematical Software, Vol. 16, pp. 1-17, 1990.

[40] I. Duff. "A survey of sparse matrix research". In Proceedings IEEE, Vol. 65, pp. 500-535, 1977.

[41] R. Eickemeyer, R. Johnson, S. Kunkel, M. Squillante and S. Liu. "Evaluation of multithreaded uniprocessors for commercial application environments". In Proceedings of the 23th Annual International Symposium on Computer Architecture, pp. 203-212, May 1996.

[42] K. Esseghir. "Improving data locality for caches". Master's thesis, Department of Computer Science, Rice University, 1993.

[43] K. I. Farkas, N. P. Jouppi and P. Chow. "How useful are non-blocking loads, stream buffers and speculative execution in multiple issue processors". In Proceedings of the 1st IEEE Symposium on High-Performance Computer Architecture, pp. 78-89, Los Alamitos, California, 1995.

[44] A. Fernández. "Systematic transformation of systolic algorithms for programming multicomputers" (in Spanish). Ph.D. Thesis, Universitat Politècnica de Catalunya, 1992.

[45] A. Fernández, J.M. Llabería, M. Valero-García. "Loop transformation using non-unimodular matrices". IEEE Transactions on Parallel and Distributed Systems, Vol. 6, No. 8, pp. 832-840, August 1995.

[46] J. Ferrante, V. Sarkar and W. Thrash. "On estimating and enhancing cache effectiveness". In Proceedings of the 4th International Workshop on Languages and Compilers for Parallel Computing, 1991.

[47] J. Fu, J. Patel and B. Janssens. "Stride directed prefetching in scalar processors". In Proceedings of the 25th International Symposium on Microarchitecture (MICRO-25), pp. 102-110, December 1992.

[48] K. Gallivan, W. Jalby, U. Meier and A. H. Sameh. "Impact of hierarchical memory systems on linear algebra design". International Journal of Supercomputer Applications, Vol. 2, No. 1, pp. 12-48, Spring 1988.

[49] K. Gallivan, W. Jalby and D. Gannon. "On the problem of optimizing data transfers for complex memory systems". In Proceedings of the ACM International Conference on Supercomputing, ACM Press, pp.238-252, July 1988.

[50] D. Gannon, W. Jalby and K. Gallivan. "Strategies for cache and local memory management by global program transformations". In Proceedings of the 1st International Conference on Supercomputing, 1987

[51] G. Goff, K. Kennedy and C-W. Tseng. "Practical dependence testing". In Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation, pp. 15-29, June 1991.

[52] G. H. Golub and C. F. Van Loan. "Matrix computation". Johns Hopkins University Press, 1989.

[53] J.L. Hennessy and D. A. Patterson. "Computer architecture: a quantitative approach". 2nd Edition, Morgan Kaufmann, 1996.

[54] F. Irigoin. "Partitionement des boucles imbriquées: une technique d'optimisation pour les programmes scientifiques". Ph.D. Thesis, Université Paris-VI, 1987.

[55] F. Irigoin and R. Triolet. "Supernode partitioning". In Proceedings of the 15th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 319-329, January 1988.

[56] S. Jain. "Circular scheduling: a new technique to perform software pipelining". In Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, pp. 219-228, June 1991

[57] M. Jiménez, J.M. Llabería, A. Fernández and E. Morancho. "A unified transformation technique for multilevel blocking". Technical Report UPC-DAC-1995-51, Computer Architecture Department, Universitat Politècnica de Catalunya, December 1995.

[58] M. Jiménez, J.M. Llabería and A. Fernández. "Performance evaluation of tiling for the register level". In Proceedings of the 4th International Symposium on High-Performance Computer Architecture, January 1998.

[59] M. Jiménez, J.M. Llabería and A. Fernández. "Loop bounds computation for multilevel tiling". In Proceedings of the 6th Euromicro Workshop on Parallel and Distributed Processing, pp. 445-452, January 1998.

[60] M. Jiménez, J.M. Llabería, A. Fernández and E. Morancho. "A general algorithm for tiling the register level". In Proceedings of the 12th ACM International Conference on Supercomputing, July 1998.

[61] N. P. Jouppi. "Cache write policies and performance". In Proceedings of the 20th Annual International Symposium on Computer Architecture, pp. 191-201, San Diego, California, May 1993.

[62] N. P. Jouppi. "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers". In Proceedings of the 17th Annual International Symposium on Computer Architecture, pp. 364-373, May 1990.

[63] Y. Ju and H. Dietz. "Reduction of cache coherence overhead by compiler data layout and loop transformation". In 4th International Workshop on Languages and Compilers for Parallel Computing, pp. 344-358, August 1991.

[64] T. Juan, T. Lang and J.J. Navarro. "The difference-bit cache". In Proceedings of the 23th Annual International Symposium on Computer Architecture, pp. 114-120, May 1996.

[65] M. Kandemir, J. Ramanujam and A. Choudhary. "A compiler algorithm for optimizing locality in loop nests". International Conference on Supercomputing, pp. 269-271, May 1997.

[66] M. Kandemir, A. Choudhary, J. Ramanujam and P. Banerjee. "Improving locality using loop and data transformations in a integrated framework". In Proceedings of the 31th International Symposium on Microarchitecture (MICRO-31), December 1998.

[67] M. Kandemir, A. Choudhary, J. Ramanujam and M. Kandaswamy. "Locality optimization algorithms for compilation of out-of-core codes". In Journal of Information science and Engineering, Vol. 14, No. 1, pp. 107-138, March 1998.

[68] J. Keller. "The 21264: a superscalar Alpha processor with out-of-order execution". In Microprocessor Forum, October 1996.

[69] K. Kennedy and K. McKinley. "Optimizing for parallelism and data locality". International Conference on Supercomputing, pp. 323-334, July 1992.

[70] K. Mc Kinley, S. Carr and C-W. Tseng. "Improving locality with loop transformations". ACM Transactions on Programming Languages and Systems, Vol. 18, No. 4, pp. 424-453, July 1996.

[71] K. Mc Kinley and O. Temam. "A quantitative analysis of loop nest locality". In the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, October 1996.

[72] A. Klaiber and H. Levy. "An architecture for software-controlled data prefetching". In Proceedings of the 18th Annual International Symposium on Computer Architecture, pp. 43-63, May 1991.

[73] P. Knijenburg and A. Bik. "On reducing overhead in loops". In Proceedings of the 5th International Workshop on Compilers for Parallel Computers, pp. 200-211, 1995.

[74] I. Kodukula, N. Ahmed and K. Pingali. "Data-centric multi-level blocking". In Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation, 1997.

[75] D. Kroft. "Lockup-free instruction fetch/prefetch cache organization". In Proceedings of the 8th Annual International Symposium on Computer Architecture, pp. 81-85, 1981.

[76] D. Kuck. "Dependence graphs and compiler optimizations". In the 8th Annual ACM Symposium on Principles of Programming Languages, pp. 207-218, 1981.

[77] R. H. Kuhn. "Optimization and interconnection complexity for: parallel processors, single-stage networks, and decision trees". Ph.D. Thesis, University of Illinois at Urbana-Champaign, February 1980.

[78] R. H. Kuhn, B. Leasure and S. M. Shah. "The KAP parallelizer for DEC Fortran and DEC C programs". Digital Technical Journal, Vol. 6, No. 3, 1994.

[79] M. Lam. "Software pipelining: an effective scheduling technique for VLIW machines". In Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation, pp. 318-328, June 1988.

[80] M. Lam, E.Rothberg and M. Wolf. "The cache performance and optimizations of blocked algorithms". International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 63-74, 1991.

[81] J. Laudon and D. Lenoski. "The SGI Origin: a ccNUMA highly scalable server". In Proceedings of the 24th Annual International Symposium on Computer Architecture, pp. 241-251, July 1997.

[82] A. Lebeck and D. Wood. "Cache profiling and the SPEC benchmarks: a case study". IEEE Computer, Vol. 27, No. 10, October 1994.

[83] W. Li. "Compiler cache optimizations for banded matrix problems". In Proceedings of the 1995 International Conference on Supercomputing, pp. 21-30, July 1995.

[84] W. Li and K. Pingali. "Access normalization: loop restructuring for NUMA computers". ACM Transaction on Computer Systems, Vol. 11, No. 4, pp. 353-375, November 1993.

[85] W. Li and K. Pingali. "A singular loop transformation framework based on non-singular matrices". Technical Report TR92-1294, Department of Computer Science, Cornell University, July 1992.

[86] L-C. Lu and M. Chen. "Subdomain dependence test for massive parallelism". International Conference on Supercomputing, pp. 962-972, November 1990.

[87] L-C. Lu and M. Chen. "A new loop transformation techniques for massive parallelism". Technical Report YALEU/DCS/TR-833, Yale University, 1990.

[88] N. Manjikian and T. Abdelrahman. "Fusion of loops for parallelism and locality". IEEE Transactions on Parallel and Distributed Systems, Vol. 8, No. 2, pp. 193-209, February 1997.

[89] D. Maydan, J. Hennessy and M. Lam. "Efficient and exact data dependence analysis". In ACM SIGPLAN'91 Conference on Programming Language Design and Implementation, pp. 1-14, June 1991.

[90] N. Mitchell, L. Carter, J. Ferrante and K. Högstedt. "Quantifying the multi-level nature of tiling interactions". In Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing, August 1997.

[91] N. Mitchell, L. Carter, and J. Ferrante. "A compiler perspective on architectural evolutions". IEEE Technical Communications on Computer Architectures (TCCA), pp. 7-9, June 1997.

[92] S. Moon and R. Saavedra. "Hyperblocking: a data reorganization method to eliminate cache conflicts in tiled loop nests". Technical Report USC-CS-98-671, Computer Science Department, University of Southern California, 1998.

[93] T. C. Mowry. "Tolerating latency through software-controlled data prefetching". Ph.D. thesis, Department of Electrical Engineering, Stanford University, May 1994.

[94] T. Mowry and A. Gupta. "Tolerating latency through software-controlled prefetching in shared-memory multiprocessors". Journal of Parallel and Distributed Computing, Vol. 12, No. 2, pp. 87-106, 1991.

[95] T. Mowry, M. Lam and A. Gupta. "Design and evaluation of a compiler algorithm for prefetching". In Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems, Vol. 27, pp. 62-73, October 1992.

[96] S. S. Muchnick. "Compiler design implementation". Morgan Kaufmann Publishers, 1997.

[97] J.J Navarro, A. Juan, M. Valero, J.M. Llabería and T. Lang. "Multilevel orthogonal blocking for dense linear algebra computations". IEEE Computer Society TC on Computer Architecture Newsletter, pp. 10-14, 1993.

[98] J.J Navarro, T. Juan, T. Lang. "MOB forms: a class of multilevel block algorithms for dense linear algebra operations". International Conference on Supercomputing, pp. 354-363, July 1994.

[99] M. O'Boyle and P. Knijnenburg. "Integrating loop and data transformations for global optimization". IEEE International Conference on Parallel Architectures and Compilation Techniques, pp. 12-19, October 1998.

[100] S. Palacharla and R.E. Kessler. "Evaluating stream buffers as secondary cache replacement". In Proceedings of the 21th Annual International Symposium on Computer Architecture, pp. 24-33, Los Alamitos, California, April 1994.

[101] D. Patterson, T. Anderson and K. Yelick. "A case for intelligent DRAM: IRAM". In Hot Chips VIII, August 1996.

[102] A. K. Poterfield. "Software methods for improvement of cache performance on supercomputer applications". Ph.D. thesis, Department of Computer Science, Rice University, May 1989.

[103] W. Pugh. "A practical algorithm for exact array dependence analysis". Communications of the ACM, Vol. 35, No. 8, pp. 102-114, 1992.

[104] G. Quintana-Orti, X. Sun and C. Bischof. "A BLAS-3 version of the QR factorization with column pivoting". LAPACK Working Note #114, August 1996.

[105] J. Ramanujan. "Beyond unimodular transformations". The Journal of Supercomputing, Vol. 9, No. 4, pp. 365-389, 1995.

[106] B. Rau. "Iterative modulo scheduling". In Proceedings of the 27th International Symposium on Microarchitecture (MICRO-27), pp. 63-74, December 1994.

[107] G. Rivera and C-W. Tseng. "Data transformations for eliminating conflict misses". In Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation, 1998.

[108] R. Saavedra, W. Mao, D. Park, J. Chame and S. Moon. "The combined effectiveness of unimodular transformations, tiling and software prefetching". In the 10th International Parallel Processing Symposium (IPPS'96), April 1996.

[109] H. Samukawa. "A proposal of level 3 interface for band and skyline matrix factorization subroutines". International Conference on Supercomputing, pp. 397-406, July 1993.

[110] R. Schreiber and J. Dongarra. "Automatic blocking of nested loops". Technical Report, RIACS, NASA Ames Research Center and Oak Ridge Nat'l Laboratory, May 1990.

[111] A. Schrijver. "Theory of linear and integer programming". Wiley-Interscience series in Discrete Mathematics, John Wiley and Sons, 1986.

[112] A. Seznec. "DASC cache". In Proceedings of the 1st International Symposium on High-Performance Computer Architecture, pp.134-143, January 1995.

[113] S. Singhai and K. McKinley. "A parametrized loop fusion algorithm for improving parallelism and cache locality". The Computer Journal, Vol. 40, No. 6, pp. 340-355, 1997.

[114] A. J. Smith. "Sequential program prefetching in memory hierarchies". IEEE Computer, Vol. 11, No. 12, pp. 7-21, December 1978

[115] A. Suárez, J.M. Llabería and A. Fernández. "Scheduling partitionings in systolics algorithms". IEEE International Conference on Application-Specific Array Processors (ASAP'92), pp. 619-630, 1992.

[116] O. Temam, E. Granston and W. Jalby. "To copy or not to copy: a compile-time technique for assessing when data copying should be used to eliminate cache conflicts". In Supercomputing'93, pp. 410-419, 1993.

[117] M. Tremblay, D. Greenley and K. Normoyle. "The design of the microarchitecture of UltraSPARC-I". In Proceedings of the IEEE, Vol. 83, No. 12, pp. 1653-1663, December 1995.

[118] D. M. Tullsen, S. J. Eggers and H. M. Levy. "Simultaneous multithreading: maximizing on-chip parallelism". In Proceedings of the 23th Annual International Symposium on Computer Architecture, pp. 191-202, May 1996.

[119] S. VanderWiel and D. Lilja. "When caches aren't enough: data prefetching techniques". IEEE Computer, pp. 23-30, July 1997.

[120] J. Veenstra. "MINT3 user manual". SiliconGraphics Computer Systems, 1998

[121] M. Wolf. "Improving locality and parallelism in nested loops". Ph.D. Thesis, Stanford University, August 1992.

[122] M. Wolf and M. Lam. "A data locality optimizing algorithm". In Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation, pp. 30-44, June 1991.

[123] M. Wolf and M. Lam. "A loop transformation theory and an algorithm to maximize parallelism". IEEE Transactions on Parallel and Distributed System, Vol. 2, No. 4, pp. 452-471, October 1991.

[124] M. Wolf, D. Maydan and D.K. Chen. "Combining loop transformations considering caches and scheduling". In Proceedings of the 29th International Symposium on Microarchitecture (MICRO-29), pp. 274-286, December 1996.

[125] M. Wolfe. "Optimizing supercompilers for supercomputers". Ph.D. Thesis, University of Illinois, October 1982.

[126] M. Wolfe. "Advanced loop interchange". In Proceedings of the 1986 International Conference on Parallel Processing, August 1986.

[127] M. Wolfe. "Iteration space tiling for memory hierarchies". In Proceedings of the 3rd SIAM Conference on Parallel Processing for Scientific Computing, December 1987.

[128] M. Wolfe. "More iteration space tiling". International Conference on Supercomputing, pp. 655-664, 1989.

[129] M.Wolfe. "High performance compilers for parallel computing". Addison-Wesley Publishing Company, 1996.

[130] K. C. Yager. "The Mips R10000 superscalar microprocessor". IEEE Micro, pp. 28-40, April 1996.