A Cost-Effective Implementation of Multilevel Tiling

Marta Jiménez, José M. Llabería, and Agustín Fernández

Abstract—This paper 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 (nonrectangular loop nest). 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 (i.e., nonrectangular loop nests), 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. This paper presents an efficient implementation of multilevel tiling that computes exact loop bounds and has a much lower complexity than 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. For loop nests having very simple affine functions as bounds, results show that our method is between 1.5 and 2.8 times faster than conventional techniques. For loop nests having not so simple bounds, we have measured speedups as high as 2,300. Additionally, our technique allows eliminating redundant bounds efficiently. Results show that eliminating redundant bounds in our method is between 2.2 and 11 times faster than in conventional techniques for typical linear algebra programs.

Index Terms—Compilers, multilevel tiling, loop transformations, memory hierarchy.

1 INTRODUCTION

Tiling is a loop transformation that a compiler can use to automatically create block algorithms [1], [2], so that locality is achieved [3], [4], [5] and/or parallelism is exploited [6], [7]. It partitions the iteration space defined by the loop structures into tiles of the same size and shape. In order to exploit several levels of parallelism and/or achieve data locality in several memory levels, multilevel tiling has to be performed. Multilevel tiling consists of dividing a tile of a higher level into subtiles of the same shape. Each level of tiling exploits one level of the memory hierarchy or one level of parallelism. To implement multilevel tiling, several researchers have proposed applying tiling level by level [2], [8].

With current and future architectures having complex memory hierarchies, multiple processors, and/or multiple hardware contexts, it is quite common that the compiler has to perform tiling at four or more levels (processor-level parallelism, level-2 cache, L1-cache, thread-level parallelism, and registers). Previous work has shown that, when tiling for multiple memory levels, the register level is the most important one, more so than the cache levels [1], [2], [9], [10], [11], although for loop nests with very large working sets, tiling for various cache levels is also very important.

As an example, Fig. 1 shows the performance (in Mflop/s) obtained by the linear algebra problem STRMM (from BLAS3 [12]) when tiling for different memory levels and varying the problem size. Results show that, in modern microprocessors, it is much more important to exploit the register level than to exploit just the cache level. The reason is that 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 excessively limit processor performance. Nevertheless, best performance is achieved by tiling for both levels simultaneously because it achieves the benefits of both individual levels. We can see in Fig. 1 that tiling for both levels achieves even higher performance than hand-optimized vendor-supplied numerical libraries. In general, the compiler should perform multilevel tiling, including the register level, in order to achieve high performance [13].

When tiling is being applied at the register level, it is necessary (after dividing the original iteration space into tiles) to fully unroll the innermost loops that traverse the iterations inside the register tiles. This is achieved by applying Index Set Splitting (ISS) repeatedly on the tiled loop nest; every time ISS is applied, a loop nest is duplicated. The number of times that ISS 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) [13], [14]. 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 never be executed. Thus, when multilevel tiling includes the register level, it is critical to compute exact bounds1 and to eliminate redundant bounds,2 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 loops (i.e., nonrectangular

---

1. A loop has exact bounds if it never executes empty iterations.
2. 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 nest.
loop nests), can be extremely time consuming. Of course, for loop nests that define rectangular iteration spaces, the complexity of computing exact bounds is linear on the number of loops in the multilevel tiled code. However, this is not the case for loop nests defining nonrectangular iteration spaces. These nonrectangular loop nests are commonly found in linear algebra programs [12] or can arise as a result of applying transformations such as loop skewing. Moreover, conventional multilevel tiling techniques generate many redundant bounds. Eliminating these redundant bounds is also a very time consuming job that can increase a program’s compile time significantly [15], [16].

In this paper, we present an efficient implementation of multilevel tiling that computes exact loop bounds at a much lower complexity than traditional techniques. The main idea behind our algorithm is that we deal with all levels to be tiled simultaneously, instead of applying tiling level by level. 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. We also compare our implementation against traditional techniques for typical linear algebra codes and show that our method is between 1.5 and 2.8 times faster.

Moreover, our implementation generates fewer redundant bounds and allows removing the remaining redundant bounds in the innermost loops at a much lower cost than traditional implementations. We also compare both implementations in terms of redundant bounds generated and the cost of eliminating these redundant bounds. We show that eliminating redundant bounds with our proposal is between 2.2 and 11 times faster than in a conventional implementation. Overall, using our algorithm, tiling for the register level becomes viable even in the face of complex nonrectangular loop nests and when tiling for many levels.

The rest of this paper is organized as follows: In Section 2, we present the framework where we develop our technique. In Section 3, we briefly explain how conventional techniques implement multilevel tiling and evaluate its complexity. In Section 4, we show how multilevel tiling can be performed dealing with all levels simultaneously. In Section 5, we give our efficient implementation of multilevel tiling and evaluate its complexity. In Section 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 7, we compare our implementation against conventional techniques in terms of complexity, redundant bounds generated, and cost of eliminating redundant bounds. In Section 8, we present previous work related to multilevel tiling and exact loop bound computation and, in Section 9, we draw some conclusions.

Finally, we want to remark that, in this paper, 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.

2 Framework

The set of iterations determined by the bounds of $n$ nested loops is a convex subset of $\mathbb{Z}^n$, and we will refer to it as $BIS$ (Bounded Iteration Space):

$$BIS = \{ \vec{I} = (I_1, \ldots, I_n) | (L_1 \leq I_1 \leq U_1, \ldots, L_n \leq I_n \leq U_n) \}$$

where $\vec{I}$ is an $n$-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, and we will refer to each affine function as a simple bound.

The $BIS$ can be specified in a matrix form as follows:

$$A \cdot \vec{I} \leq \beta$$

where each row of matrix $A$ and vector $\beta$ has the coefficients of the loop iteration variables and the independent term of each lower or upper simple bound, respectively [17]. The $n$ elements of vector $\vec{I}$ are the loop iteration variables $(I_1, \ldots, I_n)$.

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

$$BTIS = \{ \vec{J} = T \cdot \vec{I} | \vec{I} \in BIS \}.$$ 

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 $BIS$:

$$A \cdot \vec{I} \leq \beta \quad \Rightarrow \quad A \cdot T^{-1} \cdot \vec{J} \leq \beta$$

The exact bounds of the $MCS$ can be extracted from the matrix inequality $A \cdot \vec{J} \leq \beta$, using the Fourier-Motzkin Elimination algorithm [16], [17].

When $T$ is unimodular, all the integer points of the $BTIS$ have an integer antimage 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 nonunimodular (its determinant is different from ±1), there are holes (integer points without integer antimage) in the $BTIS$ and, in particular, at the boundaries of the $BTIS$. In this case, to scan the $BTIS$ correctly, all these holes must be skipped. In particular, the bounds of the $MCS$ obtained through the Fourier-Motzkin algorithm must be corrected to obtain the precise bounds of the $BTIS$. 

---

Fig. 1. Performance of STRMM on an ALPHA 21264 processor, when tiling for different memory levels and varying the problem size from 10 to 1,500. TRL is the code after tiling only for the register level. TCL corresponds to tiling only for the cache level. TCRL corresponds to tiling for both cache and registers levels, and BLAS3 library is the performance obtained if we call the hand-optimized BLAS3 library to perform the operation.
Fig. 2. Steps performed when a nonunimodular transformation $T$ is applied. $HNF(T)$ is the Hermite Normal Form of $T$.

Fernández et al. [18] address the problem of correcting in a systematic way the bounds of the $MCS$ given by the Fourier-Motzkin 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 $T$ [18], [19]. Both $H$ and $T$ generate the same lattice in $\mathbb{Z}^n$. 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 $MCS$ with some special nonlinear functions that involve the nondiagonal elements of $H$ [18].

Summarizing, the result of applying a nonunimodular transformation to a convex iteration space is a nonconvex space. To obtain the precise bounds of the nonconvex 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 [19]. Fig. 2 shows a diagram of the steps performed when a nonunimodular transformation is applied.

3 CONVENTIONAL TILING IMPLEMENTATION

In this section, we briefly explain how conventional techniques implement multilevel tiling and evaluate its complexity.

3.1 Implementation

Conventional tiling techniques implement one level of tiling using two well-known transformations: strip-mining and loop permutation [7], [20]. 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 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 [2], [4]. Conventional techniques apply strip-mining and loop permutation repeatedly, as many times as dimensions have to be partitioned.

Fig. 4 shows how conventional implementations work. Strip-mining decomposes a loop 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 reorder 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 reorder the inner loops as desired for the final code. In the final tiled code, the loops that step between tiles are always the outermost loops (we refer to them as TI-loops), and the loops that traverse the points within the tiles are the innermost ones (we refer to them as EL-loops).

The loop bounds after the strip-mining transformation are directly obtained by applying the formula shown in Fig. 3 [20]. Using this formula, the tile boundaries are always parallel to the iteration space axes.

The loop bounds after a loop permutation can be obtained using the theory of unimodular transformations [7], since the loops involved in a permutation are always loops that have steps equal to 1 and, therefore, they define a convex iteration space. To compute the exact bounds, the Fourier-Motzkin algorithm is used when the unimodular permutation matrix is applied [15].

Multilevel tiling has been implemented by applying tiling level by level [2], [8], going from the highest (i.e., parallelism) to the lowest level (i.e., register level). In Fig. 4, another level of tiling can be performed by applying tiling again to loops J and I of the resulting code.

3.2 Complexity

The most expensive steps of a conventional implementation are the loop permutation transformations or, more precisely, the steps for computing the exact bounds using the Fourier-Motzkin algorithm (FM algorithm) [16], [20]; the Appendix reviews the Fourier-Motzkin algorithm and its complexity.

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 $(m - n)$ times the FM algorithm on a set of $n$ loops (the $n$ innermost loops). 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. Thus, the complexity of a conventional implementation depends doubly exponentially, not only on the number of loops involved in the permutation, but also on the number of times the FM algorithm is executed (number of TI-loops in the final code). Thus, the complexity (rounded off to the complexity of the last execution of the FM algorithm) can be expressed by the following formula:

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

4 SIMULTANEOUS MULTILEVEL TILING

In this section, we show how multilevel tiling can be implemented to deal with all levels simultaneously. The idea behind our Simultaneous MultiLevel Tiling algorithm (SMLT) consists of first, applying strip-mining to all loops at all levels (strip-mining phase) and, afterward, performing a single loop permutation transformation (loop permutation phase) to obtain the desired order of the loops. We will use a running example (Fig. 5) to illustrate the whole transformation process. Moreover, we also present a very important property of our algorithm that allows us to establish the complexity of SMLT.
we want to apply to (Nonconvex Bounded Iteration Space). At this point, NCBIS space. We will refer to this nonconvex iteration space as obtain a new loop nest that describes a nonconvex iteration transformation (I)

we will tile both dimensions (two levels of tiling: At the lowest level (i.e., register level), to which loop tiling is going to be applied. We will perform Level Tiling (SMLT) works. Fig. 5a shows the original code always applied to loops with step equal to one.

level (i.e., cache level), we will only tile dimension I (shows the code after strip-mining all loops at all levels)

III loop index identifiers (NCBIS however, the source iteration space (PBIS)

4.2 Loop Permutation Phase

The iteration space defined by the original loop nest is a convex subset of $Z^n$ and, hence, the theory of nonsingular transformations is not directly applicable. The NCBIS is nonconvex because some of the loops have nonunit steps and, therefore, there are holes in the iteration space. In [13], we described a method to obtain the transformed iteration space (BTIS) when applying a unimodular loop permutation transformation to the nonconvex space NCBIS.

The development of our work was based on the characterizations of the space NCBIS and of a general space obtained after applying a nonunimodular transformation to a convex space [18]. 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 (represented as $\{H^{-1}, oft\}$) that transforms NCBIS to a convex space. We will refer to this convex iteration space as PBIS (Previous Bounded Iteration Space). In fact, oft is an offset vector and $H^{-1}$ is a diagonal matrix that represents a seminormalization transformation [20] and makes all loops in PBIS have a step equal to one. Next, the multilevel tiled iteration space BTIS can be obtained by applying a nonunimodular transformation $T = T^p \cdot \{H, oft\}$ to PBIS. This transformation performs the loop permutation and undoes the previous seminormalization applied to NCBIS. Fig. 6 shows a diagram of the Simultaneous MultiLevel Tiling (SMLT) 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. 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.

**Definition 1.** Let $H$ and $oft$ be an $m \times m$ diagonal matrix and an $m$-dimensional column vector, respectively:

$$H = \begin{pmatrix} B_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & B_m \end{pmatrix}, \quad oft = \begin{pmatrix} oft_1 \\ \vdots \\ oft_m \end{pmatrix},$$

where $oft_k, B_k \in Z, \text{ and } 0 \leq oft_k < B_k (1 \leq k \leq m)$. 

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig4.png}
\caption{Conventional implementation of tiling. We show the order in which strip-mining and loop permutation are applied to partition both space dimensions in a 2-deep nested loop to perform one level of tiling. $B_{II}$ and $B_{II}$ are the tile sizes in each dimension.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig3.png}
\caption{Formula of strip-mining. II is the loop that iterates between strips, I is the 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.}
\end{figure}
Definition 2. The $m$-deep loop nest obtained after the strip-mining phase defines the $m$-dimensional nonconvex space $NCBIS$. A loop $I^{NCBIS}_k$ in $NCBIS$ ($1 \leq k \leq m$) can be written, in the general case, as follows:

$$
\text{do } I^{NCBIS}_k = L^{NCBIS}_k \cdot B_k, \quad B_k
$$

where

- $L^{NCBIS}_k = [(L^{NCBIS}_k - o_{ft_k})/B_k] \cdot B_k + o_{ft_k}$,

Fig. 5. (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) Minimum Convex Space of $NCBIS$, directly obtained from (b). (d) Loop permutation matrix $T^p$ used in the example. (e) Minimum Convex Space of $BTIS$, obtained using the FM algorithm. (f) Matrix $H^p$ and vector $o_{ft}^p$ used to correct the bounds of the MCS of $BTIS$. (g) Exact loop bounds of $BTIS$.
Let 

\[
L^M_{k} = L^NCBIS_{k} - B_k + 1
\]

and the bounds of the MCS of NCBIS can be written in matrix form 

\[
A^NCBIS, I^NCBIS \leq B^NCBIS
\]

where 

\[
A^NCBIS = [I_{1}^{NCBIS}, \ldots, I_{m}^{NCBIS}]^{T}
\]

are the loop iteration variables of NCBIS.

**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 \((1 \leq k \leq m)\) that traverses BTIS has the following form:

\[
I^{BTIS} = L^{BTIS}_k B^P_k, \quad U^{BTIS}_k B^P_k
\]

where

\[
L^{BTIS}_k = [I^{BTIS}_k - o\!f^P_k/B^P_k] \cdot B^P_k + o\!f^P_k,
\]

\[
L^{BTIS}_k, \quad U^{BTIS}_k \quad \text{are obtained by solving the system} \quad A^NCBIS \cdot (T^P)^{-1} \cdot I^{BTIS} \leq B^NCBIS, \quad \text{using the FM algorithm where} \quad I^{BTIS} = [I^{BTIS}_1, \ldots, I^{BTIS}_m]^{T}
\]

are the loop iteration variables of BTIS,

- \( o\!f^P_k \) is the \( k \)th element of vector \( o\!f^P \) = \( T^P \cdot o\!f \), and

\[
B^P_k \quad \text{is the} \quad k \text{th diagonal element of a matrix} \quad H^P = T^P \cdot H \cdot (T^P)^{-1}.
\]

The proofs for Lemma 1 and Theorem 1 can be found in [13].

Summarizing, to compute the bounds of BTIS in a systematic way, the following steps have to be performed:

**Step 1.** Compute matrix \( H \), vector \( o\!f \), and the bounds of the MCS of NCBIS directly from the bounds of NCBIS. Recall from Lemma 1 that \( L^M_{k} \) and \( U^M_{k} \) (1 \( \leq k \leq m \)) (the lower and upper bounds of the MCS along dimension \( I^NCBIS_{k} \), respectively) can be directly obtained from the bounds of NCBIS as follows:

\[
L^M_{k} = L^NCBIS_{k} - B_k + 1
\]

\[
U^M_{k} = U^NCBIS_{k}
\]

Then, the MCS of NCBIS can be written in matrix form as 

\[A^NCBIS, I^NCBIS \leq B^NCBIS\]

In Fig. 5c, we show matrix \( H \) and vector \( o\!f \) of our running example, and Fig. 5d shows the bounds of the MCS of NCBIS.

**Step 2.** Compute the bounds of the MCS of the BTIS. They are extracted from the matrix inequality 

\[A^NCBIS \cdot (T^P)^{-1} \cdot I^{BTIS} \leq B^NCBIS\]

using the FM algorithm (Theorem 1). \( T^P \) is the unimodular loop permutation transformation.

In Fig. 5e, we show the loop permutation transformation \( T^P \) used in our running example, and Fig. 5f shows the bounds of the MCS of BTIS.

**Step 3.** Correct the bounds of the MCS of the BTIS using vector \( o\!f^P = T^P \cdot o\!f \) and matrix \( H^P = T^P \cdot H \cdot (T^P)^{-1} \). At this point, we have obtained the exact bounds of BTIS.

Fig. 5g shows matrix \( H^P \) and vector \( o\!f^P \) of our example, and Fig. 5h shows the exact bounds of BTIS.

Finally, we want to notice that the loop body of the original iteration space (BIS) does not need to be rewritten after the SMLT 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.

### 4.3 SMLT Property

Of all the steps required to implement our multilevel tiling technique, the most expensive one is the step that computes the bounds of the MCS of the BTIS using the FM algorithm (Step 3 in Fig. 5). At this step, the FM algorithm is executed on a set of \( m \) loops \((m \gg n)\) with a total of \( p \) simple bounds \((p \gg q)\), \( n \) and \( q \) being the number of loops and simple bounds in the original loop nest. Thus, the complexity of SMLT (computed by directly applying the formula of the FM complexity (see the Appendix)) could be:

\[
C = O \left( \frac{m^2}{2} \cdot p^{(q-1)} \right)
\]

and, therefore, it would depend doubly exponentially on the number of loops in the multilevel tiled code. However, it can be demonstrated that in SMLT, the following theorem holds when computing the exact loop bounds in the transformed space.

**Theorem 2.** In SMLT, the second step of the FM algorithm (see the Appendix) does not need to be performed when the loop being solved is a TI-loop (a loop that steps between tiles).
It can be demonstrated that, when computing the bounds of a TI-loop, all new simple bounds generated by the second step of the FM algorithm are redundant and, therefore, the second step of the FM algorithm does not need to be performed. The proof can be found in [13].

This SMLT characteristic is very important because: 1) it causes the number of simple bounds not to increase quadratically when computing the bounds of TI-loops and, therefore, it allows us to demonstrate later in Section 5.3 that the complexity of SMLT does not depend doubly exponentially on the number of loops in the multilevel tiled code, and 2) it reduces the number of redundant bounds generated in the final tiled code.

5 Efficient Implementation of SMLT

In this section we propose an efficient implementation of SMLT that (together with the SMLT property) allows us to demonstrate that the whole SMLT process has a much lower complexity than traditional techniques. The main idea behind our efficient implementation is to reduce the number of simple bounds examined in each iteration of the FM algorithm executed in the third step of the SMLT process.

The rest of this section is organized as follows: First, we show how the number of simple bounds examined in each iteration of FM can be reduced, second, we summarize the SMLT algorithm and give some implementation details and, third, we analyze its complexity.

5.1 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 the permutation) and its final position (after the permutation). We note that it is not necessary to examine the bounds of other outer loops because they cannot have simple bounds that are affine function of the loop being solved. In the previous example (Fig. 5f), to compute the bounds of loop II, the FM algorithm examines the simple bounds of loops I, JJ, and J of Fig. 5d. Note that the examined loops can be contiguous TI-loops followed by their associated EL-loop (loops JJ and J in Fig. 5d).

The idea behind our efficient SMLT implementation consists in reducing the number of simple bounds examined in each iteration of the FM algorithm by representing, with a single loop (called C-loop; Cluster loop), a set of contiguous loops that are related by the strip-mining transformation. A C-loop traverses the same iteration space as the loops it is representing, but uses a fewer number of simple bounds. Then, the TI and EL-loops of the final tiled code will be stripped 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.

In the example of Fig. 5, we start with the loops in the original code I-J (Fig. 5a). These two loops are C-loops because they represent the contiguous associated TI and EL-loops of Fig. 5b. Just before computing the bounds of EL-loop I in the tiled code (Fig. 5f), it is stripped from its associated C-loop, obtaining the loop nest II-I-J (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 loops I and J.

To implement this “stripping” process, the SMLT 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, just in reverse order from what strip-mining does. Thus, we need a backward transformation of strip-mining that we will refer to as strip-clustering.

5.1.1 Strip-Clustering

Strip-clustering is a loop transformation that clusters a set of strips together. It decomposes a C-loop into two loops where the outer loop iterates between clusters of inner strips and the inner loop iterates between the inner strips. The outer loop is a new C-loop and the inner loop is the TI or EL-loop being stripped.

The strip-clustering transformation applied to a C-loop with a step equal to one is defined by the same expression as strip-mining (Fig. 3). The strip-clustering transformation applied to a C-loop with a step different from one is defined by the expressions in Fig. 7, where II is the new C-loop, II is the TI-loop being stripped, BIII and BII are their respective strip sizes (BIII > BII), and oftIII, oftII ∈ Z (0 ≤ oftIII < BIII, 0 ≤ oftII < BII) are the offsets of each strip.

Two different phases can be distinguished when strip-clustering is applied to a C-loop with a step different from one: 1) the creating phase that consists in creating the new C-loop and the TI-loop being stripped (their bounds are directly computed using the expressions of Fig. 7) and 2) the broadcasting phase that consists of modifying the simple bounds of inner loops that have simple bounds that are affine functions of the TI-loop being stripped. In the general case, the TI-loop being stripped (loop II in Fig. 7) can iterate over some points outside the tiles determined by the new C-loop.

Fig. 7. Formula for strip-clustering a C-loop with a step different from one.

4. In this context, “to strip” means extracting the innermost TI (or EL)-loop from a C-loop.
Lemma 2. Let $II \geq III$ and $II + B_{II} - 1 \leq III + B_{III} - 1$ does not always hold).

In SMLT, 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 (as well as EL-loop $I$) can have simple bounds that are affine functions of the just stripped TI-loop ($II$) and, therefore, their bounds must be modified in the broadcasting phase. The following lemma shows how these loop bounds must be modified. The proof can be found in [13].

**Lemma 2.** 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).

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

### 5.2 SMLT Algorithm

Summarizing, our implementation of SMLT consists of integrating the strip-clustering transformation inside the FM algorithm to compute the exact bounds of the $MCS$ of BTIS. It will generate the EL and TI-loops of the final multilevel tiled code, one by one, from the innermost to the outermost one. The loops of the original code are the initial C-loops and in each iteration, the SMLT 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 loop being solved is an EL-loop, the SMLT algorithm performs both steps of the FM algorithm. If the loop is a TI-loop, it only performs the first step of FM (see Section 4.3).

**Fig. 8** shows how our implementation of SMLT works using the same example of Fig. 5. It shows the order in which strip-clustering and the computation of the bounds are performed. The loop list written in the rows labeled "strip-clustering" indicates which C-loop (in bold) and which TI or EL-loop (in bold and underline) appears after strip-clustering. In rows labeled "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. In the two first iterations of SMLT, strip-clustering is applied to C-loops having steps equal to one. In the third iteration, however, strip-clustering is applied to a C-loop that has a step different from one (loop $II$) and, therefore, the creating and the broadcasting phases of strip-clustering have to be performed. In particular, the broadcasting phase must modify the bounds of loops $JJ$, $J$, and $I$.

The complete SMLT algorithm is shown in Fig. 9. List $\mathbb{L}$ 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 $\mathbb{TL}$ 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 $\mathbb{LI}$ that contains all loops in the tiled code with all the simple bounds computed. $\mathbb{LI}$ is initialized to $\mathbb{L}$ and, at the end of the algorithm, it will contain the multilevel tiled code. Other variables used in the algorithm are: 1) variable $s$ that indicates the number of loops in list $\mathbb{LI}$, 2) variable $r$ that indicates the number of loops that have already been processed, and 3) variable $\text{str.cl}$ that indicates, in each iteration, if strip-clustering has been performed or not.

List $\mathbb{TL}$ gives the order in which the loops are processed. Thus, for each of the first $m - 1$ iteration variables in $\mathbb{TL}$, the algorithm begins finding the position $j$ in $\mathbb{LI}$ where the associated C-loop of the TI or EL-loop we want to deal with is. This C-loop is then assigned to $\mathbb{AL}$ (Active Loop) and removed from $\mathbb{LI}$. Next, strip-clustering is applied to $\mathbb{AL}$, 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 $\mathbb{LI}$ in position $j$ and the TI or EL-loop is assigned to $\mathbb{AL}$. Thereafter, the SMLT algorithm computes the exact bounds of $\mathbb{AL}$ (the just stripped TI or EL-loop).

5. The bounds of the outermost loop are obtained directly after the SMLT algorithm has finished.
Finally, loop AL is inserted in LI in position r (the innermost position).

5.3 Complexity of the SMLT 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 tiled code.

With the given implementation of SMLT, the total number of simple bounds increases quadratically in each iteration only when computing the bounds of the n inner EL-loops. Therefore, the complexity of computing the bounds of all n EL-loops is:

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

and the total number of simple bounds in the n outer C-loops is, in the worst case: $\hat{q} = (q/2)^{2n-1}$.

Then, when computing the bounds of the TI-loops, the total number of simple bounds does not increase any more, because the second step of the FM algorithm does not need to be performed for these loops (Theorem 2 of Section 4.3) and the broadcasting phase does not increase the number of simple bounds of the not-yet-processed loops. Therefore, the outer C-loops together will always have less than $\hat{q}$ simple bounds.

Thus, the FM algorithm examines, at most, $\hat{q}$ simple bounds for each TI-loop being solved and the complexity of computing the bounds of all TI-loops is:

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

The complexity of SMLT is then the sum of the above stated complexities, that is:

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

Notice that the complexity of SMLT 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, this complexity is proportional to the complexity of performing a loop permutation in the original loop nest.

6 Redundant Bounds

Another problem related to the generation of multilevel tiled loop nests is the generation of redundant bounds. 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 [14] because the number of times that ISS has to be performed and the amount of code generated depend polynomially on the number of bounds the innermost loops have after tiling [13]. Every time ISS is applied, 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.

The FM algorithm used to compute the exact loop bounds can generate redundant bounds in the transformed loop nest [15], [16]. There are two kinds of redundant bounds: 1) trivial redundant bounds, whose redundancy can be deduced by only looking at the other simple bounds of the loop (these bounds can be eliminated as soon as they appear in an easy way) and 2) nontrivial redundant bounds, whose redundancy is deduced by looking at the bounds of other outer loops. To eliminate nontrivial redundant bounds, several researchers [15], [16], [23] propose the use of the FM algorithm. 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 bound is redundant and can be eliminated (to check if a system is inconsistent, the FM algorithm is used). Thus, the complexity of eliminating redundant bounds is the complexity of executing the FM algorithm as many times as bounds have to be checked. Obviously, this technique, called Exact Simplification in [16], is very time-consuming.

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 reducing the cost of eliminating the remaining redundant bounds in the innermost loops when the exact simplification technique is used.

### 6.1 Generating Fewer Redundant Bounds

Our multilevel tiling implementation reduces the number of nontrivial redundant bounds generated with respect to a traditional implementation. Moreover, our implementation allows reducing the cost of eliminating the remaining redundant bounds in the innermost loops using the exact simplification technique.

SMLT generates fewer redundant bounds than conventional techniques due to two reasons. On one hand, the second step of the FM algorithm is not performed when computing the bounds of a TI-loop because all new simple bounds generated will be redundant. Conventional implementations, however, generate these redundant bounds [13]. On the other hand, conventional implementations generate nontrivial redundant bounds when, in a loop permutation, an EL-loop $J$ is moved inside another EL-loop $J$ that has been previously strip-mined and $J$ has simple bounds that are affine functions of $I$. Moreover, these redundant bounds could be propagated to other loops in later loop permutations [13]. By contrast, SMLT does not generate these nontrivial redundant bounds, because it always computes the bounds of an EL-loop $I$ before applying strip-clustering to the outer not-yet-processed EL-loops (i.e., loop $J$).

As an example, Fig. 10 shows the code generated by SMLT and by a conventional implementation after applying one level of tiling to the skewed SOR code [2]. As it can be seen, SMLT do not generate any redundant bound (Fig. 10b) while a conventional implementation generates 10 redundant bounds (Fig. 10c).

### 6.2 Reducing the Cost of Eliminating the Remaining Redundant Bounds

SMLT reduces the cost of eliminating the remaining redundant bounds in the innermost loops (with respect to conventional implementations) for two reasons. On one hand, as just mentioned, it generates fewer redundant bounds and, therefore, fewer bounds have to be checked for
redundancy. On the other hand, the processing order of the loops in SMLT (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 and bounds involved in each FM algorithm executed by the Exact Simplification phase 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. Therefore, redundant bounds in the innermost loops cannot be eliminated until the multilevel tiling process has been finished.

In conventional techniques, there are two alternatives to eliminate the redundant bounds: 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 the Exact Simplification phase several times, it is faster than the first one because the complexity of the FM 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.

Fig. 11 illustrates, with the same example of Fig. 5, when the Exact Simplification phase is performed in both SMLT and conventional implementations. Although the number of loops involved in the executions of FM is larger in SMLT, the complexity of the overall process of eliminating the redundant bounds is smaller in SMLT than in a conventional implementation. The reasons are that 1) the number of bounds to be checked for redundancy is much larger in a conventional implementation, and 2) SMLT performs the Exact Elimination phase only once while conventional implementations perform it several times.

7 SMLT versus Conventional Tiling

In this section, we compare SMLT against conventional multilevel tiling techniques in terms of complexity, redundant bounds generated, and cost of eliminating redundant bounds. For that purpose, we have implemented both SMLT 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 nontrivial. We have also implemented the Exact Simplification technique for measuring 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 LU, Cholesky, QR, Blas3 routines, etc. These loop nests are 3-deep and have six simple bounds with only one of them being an affine function of one surrounding loop iteration variable. The measures were taken on a workstation with a SuperSPARC at 155 Mhz.

7.1 Complexity

Let us 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. Both the worst-case complexity of conventional techniques and of SMLT can be represented by a unique function, namely, \( f(X, Y) = Y \cdot (q/2)^{2X} \), where \( q \) is the number of simple bounds in the original code. Using this formula, the complexity of SMLT is given by \( f(n - 1, m - n) \) and the complexity of a conventional implementation is given by \( f((m - n) \cdot (n - 1), 1) \), 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 six simple bounds, and we perform three levels of tiling, obtaining an 8-deep tiled loop nest (that is, \( n = 3 \), \( q = 6 \), and \( m = 8 \)). For
this particular example and in the worst case, the complexity value of SMLT is \( f(2, 5) \), while that of a conventional implementation is \( f(10, 1) \). In Fig. 12, we have plotted the curves \( f(X, 1) \) and \( f(X, 5) \), with \( q \) fixed to six (note that they are overlapped), and we have also marked with points the complexity values of SMLT and a conventional implementation for our particular example.

With this figure, we want to outline that, even for a 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 three levels can yield an 8 or 9-deep loop nest. Therefore, even for small values of \( n \), SMLT can perform much better than conventional techniques.

Loop nests found, in practice, hardly ever incur in the worst-case complexity. However, we will show next that, for typical linear algebra programs, SMLT significantly improves upon conventional implementations. We have measured the compile time required by each implementation for our benchmark programs. Fig. 13 shows the average compile time required by each technique, varying the levels of tiling. Above the gray bars, we show the average increment percentage of compile time of a conventional implementation over SMLT.

Moreover, one of the simple bounds is 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 SMLT is 3.17 ms for one level and 13.8 ms for four levels. In this case, the increment percentage of compile time of the conventional implementation over SMLT varies from 215 percent for one level, to over 200,000 percent 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 [24], or can arise as a result of applying transformations such as loop skewing [20].

### 7.2 Redundant Bounds

Let us now present some data showing the number of redundant bounds generated by SMLT and by conventional techniques, and the cost of eliminating these redundant bounds in the innermost loops using the Exact Simplification technique.

First, we measured the total number of simple bounds generated by SMLT and conventional implementations. Both implementations compute exact bounds, which means that, if we eliminate all redundant bounds, both implementations obtain the same final code. Fig. 14 shows the average number of simple bounds in the final tiled code for the 12 programs, varying the levels of tiling from one to four. We also show the average number of bounds if we eliminate all redundant bounds in the codes and, above the bars, we indicate, for both implementations, the average percentage of redundant bounds over the total number of bounds generated. It can be seen that SMLT always generates fewer simple bounds than a conventional implementation. A conventional implementation generates around 14 percent of redundant bounds, while SMLT only generates around 3.5 percent.

Again, we want to note that, for more complex loop nests such as SYR2K, the number of generated bounds in a conventional implementation can exploit in an exponential manner. For the SYR2K example, the conventional implementation generates 61 simple bounds for one level of tiling and 2,511 for four levels, while SMLT 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 percent for one level of tiling to 94.7 percent for four levels. However, using the SMLT implementation, there is not any redundant bound in the tiled code. Thus, for certain classes of loop nests, the number of redundant bounds generated in a conventional implementation can increase significantly.

Second, we measured the compile time required to eliminate these redundant bounds using the Exact Simplification technique in both implementations. For that purpose, we integrated the Exact Simplification technique in both implementations and we measured the compile time required to both compute the bounds and eliminate the redundant bounds of the innermost loops.

Fig. 15 shows the average compile time required by each new implementation (SMLT+ES and Conv+ES), varying the levels of tiling from 1 to 4. Above each gray bar, we show the average increment percentage of compile time of Conv+ES over SMLT. It can be seen that SMLT+ES is significantly better than conventional techniques (SMLT+ES is between 2.2 and 11 times faster than Conv+ES) and, again, SMLT+ES behaves linearly with respect to the number of tile levels, while the conventional technique has an exponential behavior. Thus, SMLT 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.

8 RELATED WORK

There has been much research regarding loop tiling [1], [3], [5], [8] 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 shown. 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 [8], [2], making these techniques have very large complexities. In this paper, we have presented an implementation of multilevel tiling that deals with all levels simultaneously and has a lower complexity.

Wolf and Lam [7] 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 [14], redundant bounds produce a code explosion that can even prevent actual generation of the final code.

Ancourt and Irigoin [23] 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 [13], [14]. To eliminate redundant bounds, several researchers [15], [16], [23] propose the use of the FM algorithm to check if a simple bound is redundant with respect to the possible values of outer loops. This technique is very time-consuming and could be not feasible to be implemented in a compiler. Our implementation of SMLT generates fewer 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.

Moreover, Bik and Wijshoff [16] and Amarasinghe [15] 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
technique to reduce the number of bounds in the nest and, therefore, to reduce the cost of this phase.

Finally, we note that Kodukula et al. [11], [26] present a novel framework for generating blocked codes. Unlike loop tiling which is based on reasoning about the control flow of programs, their techniques are based on reasoning directly about the flow of data through the memory hierarchy. Their work is focused on exploiting the cache level and does not deal with the register level. To generate the transformed optimized code, Kodukula et al. [11], [26] use standard integer linear programming tools (such as the Omega calculator [26]) that are based on the Fourier-Motzkin algorithm. It is unclear how the authors perform multilevel blocking and generate the transformed multilevel blocked code.

9 CONCLUSIONS

To improve the performance of a program, the compiler should apply multilevel tiling to maximize the effectiveness of the memory hierarchy and/or to exploit different levels of parallelism (processors and threads). 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 nonrectangular loop nests and when tiling for several levels. Thus, to date, the drawback of generating exact loop bounds and eliminating redundant bounds was that all techniques known were extremely expensive and, therefore, difficult to integrate in a production compiler.

In this paper, we have presented a new algorithm (SMLT) 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.

We have also measured the compile time consumed by conventional techniques and by SMLT. We have shown that SMLT is between 1.5 and 2.8 times faster than conventional implementations for simple loop nests, but it can be even 2,300 times faster for more complex loop nests. Moreover, our implementation generates fewer redundant bounds and allows eliminating the remaining redundant bounds efficiently, an important issue if the register level is being exploited [13]. Overall, the efficiency of SMLT makes it possible to integrate multilevel tiling including the register level in a production compiler without having to worry about compilation time.

APPENDIX

FOURIER-MOTZKIN ELIMINATION ALGORITHM

Let \(lp\) and \(bd\) be the number of loops and simple bounds in a loop nest before applying a unimodular transformation, respectively. The Fourier-Motzkin algorithm is an algorithm that iterates \(lp - 1\) times and bounds each of the loop iteration variables of the transformed code from innermost to outermost. In each iteration, two different phases are performed. In the first phase (steps 1, 2, and 3), the 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 phase (steps 4 and 5), each of the lower simple bounds of the iteration variable solved in the first phase 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.

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

1. 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: \(A_+ \cdot x \leq \beta_+\), \(A_- \cdot x \leq \beta_-\) and \(A_0 \cdot x \leq \beta_0\), where matrices \(A_+, A_-\), and \(A_0\) have 1, -1, and 0, respectively, in the columns associated with the iteration variable to be bounded.

2. 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, \(A_-\) and \(A_0\) are defined.

3. The bounds for \(x_j\) are:

\[\max((\tilde{A}_- \cdot \tilde{x}) - \beta_-) \leq x_j \leq \min(\beta_+ - (\tilde{A}_+ \cdot \tilde{x})).\]

The bounds expressed in this way may take noninteger 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.

4. From the bounds of \(x_j\) we build the following matrix inequality:

\[\max((\tilde{A}_- \cdot \tilde{x}) - \beta_-) \leq x_j \leq \min(\beta_+ - (\tilde{A}_+ \cdot \tilde{x})).\]

5. We now take each row on the left-hand of the new matrix inequality 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 \(A_0 \cdot \tilde{x} \leq \beta_0\) and removing the redundant inequalities, form the new system \(\tilde{A} \cdot \tilde{x} \leq \tilde{\beta}\).

Steps (1) to (5) are repeated to obtain the bounds for the next iteration variables, until vector \(\tilde{x}\) has a single component.
Complexity. 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 phase of FM, the loop iteration variable can have $(bd/2)^2$ lower simple bounds and $(bd/2)^2$ upper simple bounds. Comparing 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 $(bd^2/8)^2$ new inequalities in the second iteration of FM. Therefore, since all $bd$ simple bounds could potentially involve all $lp$ iteration variables, the complexity of the FM algorithm is:

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

Thus, the complexity of FM depends doubly exponentially on the number of loops involved in the loop transformation.

ACKNOWLEDGMENTS

This work was supported by the Ministry of Education and Science of Spain (CICYT TIC2001-0995-C02-01).

REFERENCES


Marta Jiménez received the MS and the PhD degrees in computer science from the Universitat Politècnica de Catalunya (UPC) in 1992 and 1999, respectively. She is an assistant professor in the Computer Architecture Department at UPC, in Barcelona, Spain. Her research interests are memory hierarchy, parallel computer architecture, and compiler technology.

José M. Llaberia received the MS degree in telecommunication, and the MS and the PhD degrees in computer science from the Universitat Politècnica de Catalunya (UPC) in 1980, 1982, and 1983, respectively. He is a full professor in the Computer Architecture Department at UPC, in Barcelona, Spain. His research interests include processor micro-architecture, memory hierarchy, parallel computer architecture, vector processors, and compiler technology for these processors.

Agustín Fernández received the MS and the PhD degrees in computer science from the Universitat Politècnica de Catalunya (UPC) in 1988 and 1992, respectively. He is an assistant professor in the Computer Architecture Department at UPC, in Barcelona, Spain. His research interests are memory hierarchy, parallel computer architecture, and compiler technology.

For more information on this or any other computing topic, please visit our Digital Library at http://computer.org/publications/dlib.