An Efficient Code Generation Technique for Tiled Iteration Spaces

Georgios Goumas, Maria Athanasaki and Nectarios Koziris

National Technical University of Athens

School of Electrical and Computer Engineering

Computing Systems Laboratory

Zografou Campus, Zografou 15773, Athens, Greece

e-mail: {goumas, maria, nkoziris}@cslab.ece.ntua.gr

Abstract

This paper presents a novel approach for the problem of generating tiled code for nested for-loops, transformed by a tiling transformation. Tiling or supernode transformation has been widely used to improve locality in multi-level memory hierarchies, as well as to efficiently execute loops onto parallel architectures. However, automatic code generation for tiled loops can be a very complex compiler work, especially when non-rectangular tile shapes and iteration space bounds are concerned. Our method considerably enhances previous work on rewriting tiled loops, by considering parallelepiped tiles and arbitrary iteration space shapes. In order to generate tiled code, we first enumerate all tiles containing points within the iteration space and second sweep all points within each tile. For the first subproblem,

we refine upon previous results concerning the computation of new loop bounds of an iteration space that has been transformed by a non-unimodular transformation. For the second subproblem, we transform the initial parallelepiped tile into a rectangular one, in order to generate efficient code with the aid of a non-unimodular transformation matrix and its Hermite Normal Form (HNF). Experimental results show that the proposed method significantly accelerates the compilation process and generates much more efficient code.

Index Terms – Loop tiling, supernodes, non-unimodular transformations, Fourier-Motzkin elimination, code generation.

1 Introduction

Tiling or supernode partitioning has been widely used to improve locality in multi-level memory hierarchies, as well as to efficiently execute loops onto distributed memory architectures. Supernode partitioning of the iteration space was first proposed by Irigoin and Triolet in [21]. They introduced the initial model of loop tiling and gave conditions for a tiling transformation to be valid. Tiles are required to be atomic, identical, bounded and their union to span the initial iteration space. They also pointed out the two major directions for the application of tiling transformation: data locality and coarse grain parallelism. As far as tiling for locality is concerned, extensive theoretical and experimental research has been conducted and, as a result, many of the proposed techniques have already been incorporated in research and commercial compilers. In general, previous work on data locality focuses on the combination of tiling with a sequence of unimodular loop transformations (permutation, reversal, skewing etc.), in order to better exploit cache reuse [25], [36].

When executing nested loops on parallel architectures, the key issue in loop partitioning to different

processors is to mitigate communication overhead by efficiently controlling the computation to communication ratio. In distributed memory machines, explicit message passing incurs extra time overhead due to message startup latencies and data transfer delays. In order to eliminate the communication overhead, Shang [29], Hollander [10] and others, have presented methods for dividing the iteration space into independent sets of iterations, which are assigned to different processors. If the rank of the dependence vector matrix is $n_d < n$, the n-dimensional loop can always be transformed to have a maximum of $n - n_d$ outermost `DOALL` loops [37]. However, in many cases, independent partitioning of the iteration space is not feasible, thus data exchanges between processors impose additional communication delays. When fine grain parallelism is concerned, several methods have been proposed to group together neighboring chains of iterations [24], [30], while preserving the optimal hyperplane schedule [11], [31], [35].

When tiling to force coarse-grain parallelism, neighboring iteration points are grouped together to build a larger computation node, which is executed by a processor. Tiles have much larger granularity than single iterations, thus reducing synchronization points and alleviating overall communication overhead. Data exchanges are grouped and performed within a single message for each neighboring processor, before and after each atomic tile execution. In this case, scientific research focuses on determining efficient scheduling schemes [15], [16], optimal tile sizes [5], [16], [40] and optimal tile shapes. All three above mentioned factors (scheduling scheme, tile size and tile shape) greatly affect the overall completion time of a tiled algorithm. For example, small tile sizes provide more parallelism, but cause more frequent communication, while large tile sizes reduce parallelism, but also reduce the communication overhead. On the other hand, overlapping scheduling schemes can significantly reduce the overall completion time of a tiled iteration space by allowing simultaneous computation and communication phases [15], [32]. In this paper, we are mainly interested in the effect of the tile shape on the

performance of a tiled algorithm and thus, we will proceed with a more detailed discussion of related work.

There are two main reasons why one should choose between tile shapes. The first reason is that different tile shapes cause different communication volumes per tile. In this case, researchers have tried to define and calculate the communication-minimal tile shape. In their paper, Ramanujam and Sadayappan [28] gave a linear programming formulation for the problem of finding optimal tile shapes that minimize communication. Boulet et al. in [7] and Xue in [39] used a communication function that has to be minimized by linear programming approaches as well. They showed that the communication-minimal tile shape is equivalent to the shape of the algorithm's tiling cone. More importantly, the tile shape also greatly affects the overall completion time of an algorithm. In [9] and [18] the authors present analytical expressions of the idle time of a processor for 2-dimensional tiled spaces. This idle time is either the time a processor is waiting for data from another processor, or the time spent by a processor at a barrier waiting for other processors to accomplish their tasks. It is shown that the idle time depends on the rise - a parameter that relates the shape of the tile to the shape of the iteration space. Hodzic and Shang in [16] discussed the effect of the tile shape and size on the overall completion time of an algorithm, taking into account the iteration space bounds. In [17], they proved that the scheduling-optimal tile shape, i.e. the one that leads to the minimum execution time, is derived from the algorithm's tiling cone. Hogstedt et al. in [19], extend their work from [18] to more deeply nested loops and also affirm that the vectors forming the basic tile shape should be taken from the surface of the tiling cone. This means that if we properly scale n vectors taken from the surface of the tiling cone of an algorithm, according to the bounds of the iteration space, we can simultaneously obtain scheduling-optimal and communication-minimal tiling. Quite recently, Hogstedt et al. have proven in [20], that some more tile

shapes may be scheduling-optimal, according to the iteration space shape.

Despite this extensive research on the effect of the tile shape on the performance of a tiled algorithm, research or commercial parallelizing compilers do not use general parallelepiped (arbitrary) tiling [1], [3], [8], [13], [33]. In general, the parallelizing compiler community has been pessimistic about applying such transformations, due to the additional overhead to generate code for arbitrarily tiled iteration spaces and, more importantly, due to the additional overhead incorporated to the generated code itself (extra expressions are needed in the loop boundaries, in order to access iteration points within the non-rectangular tiles). However, the problem of generating code for arbitrarily tiled iteration spaces, was tackled by Ancourt and Irigoin in [4]. In their paper, the problem of calculating the exact transformed loop bounds is formulated as a large system of linear inequalities. These inequalities are formed first to calculate the exact bounds for every tile execution and second to access all iterations inside every tile. The authors use the Fourier-Motzkin elimination method to transform the above systems of inequalities so that they can be used in order to calculate the bounds of a nested loop. Unfortunately, due to the fact that the generated systems are unnecessarily large, and that the Fourier-Motzkin method is extremely complex, the proposed method results in being quite inefficient, in terms of both compilation time and quality of generated code. This means that the overhead to access the iteration points within non-rectangular tiles outweighs their theoretical gain.

In this paper, we present an efficient method to generate code for tiled iteration spaces, considering both non-rectangular tiles and non-rectangular iteration spaces. Our goal is to simplify the compilation process and to produce as efficient code as possible. We divide the main problem into the subproblems of enumerating the tiles of the iteration space and of sweeping the internal points of every tile (as in [4]). For the first problem, we continue previous work concerning the computation of loop bounds which tra-

verse an iteration space that has been transformed by a non-unimodular transformation [26], [27]. Tiling was used as an example to compute loop bounds, but the method proposed fails to enumerate all tile origins exactly. We adjust this method to access all tiles. As far as sweeping the internal points of every tile is concerned, we propose a novel method which uses the properties of non-unimodular transformations. This method is based on the observation that tiles are identical and that large computational overhead arises when non-rectangular tiles are involved. To handle this fact, we first transform the parallelepiped (non-rectangular) tile (Tile Iteration Space - TIS) into a rectangular one, then sweep the derived Transformed Tile Iteration Space ($TTIS$) and use the inverse transformation in order to access the original points. The results are adjusted in order to sweep the internal points of all tiles, taking into consideration the original iteration space bounds. We, thus, exploit the regularity of rectangle shaped tiles to produce more efficient code. In both subproblems, the resulting systems of inequalities are eliminated using Fourier-Motzkin elimination. Compared to the method presented by Ancourt and Irigoin in [4], our method outperforms in terms of efficiency. Experimental results show that the procedure of generating tiled code is greatly accelerated, since the derived systems of inequalities in our case are smaller. In addition, the generated code is much more efficient, since it contains less expressions and avoids heavy loop bound calculations imposed by non-rectangular tiles.

Note that the parallelization process of an arbitrarily tiled algorithm involves two separate tasks: the generation of the sequential tiled code and the parallelization of this code. This paper deals with the first task. Our goal is to generate efficient sequential tiled code with a low compilation time. However, since our method also simplifies the parallelization process, we will discuss some issues of the latter task as well. Nevertheless, parallelization details are beyond the scope of this paper. Xue in [34] presents a complete approach to parallelize tiled iteration spaces (sequential tiled code generation and parallelization)

but his method is restricted to rectangular tiles.

The rest of the paper is organized as follows: Basic terminology used throughout the paper and definitions from linear algebra are introduced in Section 2. We present tiling or supernode transformation in Section 3. Our method for generating tiled code is presented in detail in Section 4. In Section 5 we discuss some parallelization aspects of our generated code. In Section 6 we compare our method with the one presented in [4] and present experimental results for both compilation and run times. Finally, in Section 7 we conclude by summarizing our results.

2 Preliminaries

2.1 The Model of the Algorithms - Notation

In this paper, we consider algorithms with perfectly nested FOR-loops. That is, our algorithms are of the form:

```
FOR (j_1 = l_1; j_1 <= u_1)
   FOR (j_2 = l_2; j_2 <= u_2)
      ...
      FOR (j_n = l_n; j_n <= u_n)
         Loop Body
      ENDFOR
      ...
   ENDFOR
ENDFOR
```

where l_1 and u_1 are rational-valued parameters, l_k and u_k ($k = 2, \ldots, n$) are of the form: $l_k = max(\lceil f_{k1}(j_1, \ldots, j_{k-1}) \rceil, \ldots, \lceil f_{kr}(j_1, \ldots, j_{k-1}) \rceil)$ and $u_k = min(\lfloor g_{k1}(j_1, \ldots, j_{k-1}) \rfloor, \ldots, \lfloor g_{kr}(j_1, \ldots, j_{k-1}) \rfloor)$, where f_{ki} and g_{ki} are affine functions. Therefore, we are not only dealing with rectangular iteration spaces, but also with more general convex spaces, with the only assumption that the iteration space is defined as the bisection of a finite number of semi-spaces of the n-dimensional space Z^n.

Throughout this paper, the following notation is used: N is the set of naturals, Z is the set of integers and n is the number of nested FOR-loops of the algorithm. $J^n \subset Z^n$ is the set of indexes, or the *iteration space* of an algorithm: $J^n = \{j(j_1, ..., j_n) | j_i \in Z \wedge l_i \leq j_i \leq u_i, 1 \leq i \leq n\}$. Each point in this n-dimensional integer space is a distinct instantiation of the loop body. The iteration space J^n can also be described with a system of linear inequalities. An inequality of this system expresses a boundary surface of the iteration space. Thus, J^n can be equivalently defined as: $J^n = \{j \in Z^n | Bj \leq \vec{b}\}$. Matrix B and vector $\vec{b}$ can be easily derived from the affine functions l_k and u_k and vice versa. If A is a matrix, we denote a_{ij} the matrix element in the i-th row and j-th column and a_k the k-th row or column of A, according to the context.

2.2 Linear Algebra Concepts

We present some basic linear algebra concepts which are used in the following sections:

Definition 1 *A square matrix A is unimodular if it is integral and its determinant equals to* ± 1.

Unimodular transformations have a very useful property: their inverse transformation is integral as well. On the other hand the inverse of a non-unimodular matrix is not integral, which causes the transformed space to have "holes". We call *holes* the integer points of the transformed space that have no integer anti-image in the original space.

Definition 2 *Let A be an $m \times n$ integer matrix. We call the set $\mathcal{L}(A) = \{y | y = Ax \wedge x \in Z^n\}$ the lattice that is generated by the columns of A.*

Consequently, we can define the holes of a non-unimodular transformation as follows: if T is a non-unimodular transformation, we call *holes* the points $j' \in Z^n$, such that $T^{-1}j' \notin Z^n$. On the contrary,

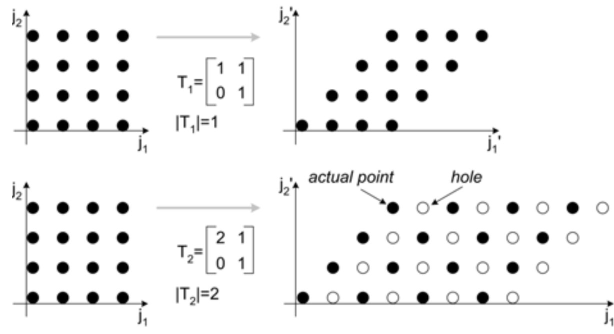

Figure 1. Unimodular and Non-Unimodular Transformations

we call *actual* points of a non-unimodular transformation T the points $j' \in Z^n$, for which it holds $T^{-1}j' \in Z^n \Leftrightarrow j' \in \mathcal{L}(T)$. Figure 1 shows the image of an iteration space after the application of a unimodular and a non-unimodular transformation. Holes are depicted with white dots and actual points with black ones.

Theorem 1 *If T is a $m \times n$ integer matrix, and C is an $n \times n$ unimodular matrix, then $\mathcal{L}(T) = \mathcal{L}(TC)$.*

Proof: *Given in [26].*

Definition 3 *We say that a square, non-singular matrix $H = [\vec{h_1}, \dots, \vec{h_n}] \in R^{n \times n}$ is in Column Hermite Normal Form (HNF) iff H is lower triangular ($h_{ij} \neq 0$ implies $i \geq j$) and for all $i > j$, $0 \leq h_{ij} < h_{ii}$ (the diagonal is the greatest element in the row and all entries are positive.)*

Theorem 2 *If T is a $m \times n$ integer matrix of full row rank, then there exists an $n \times n$ unimodular matrix C such that $TC = [\widetilde{T}0]$ and $\widetilde{T}$ is in Hermite Normal Form.*

Proof: *Given in [26].*

Every integer matrix with full row rank has a unique Hermite Normal Form. By Theorem 1, we conclude that $\mathcal{L}(T) = \mathcal{L}(\widetilde{T})$ which means that an integer matrix of full row rank and its HNF produce the same lattice. This property is very useful for code generation of tiled spaces.

2.3 Fourier-Motzkin Elimination Method

The Fourier-Motzkin elimination method (FME) can be used to convert a system of linear inequalities $A\vec{x} \leq \vec{a}$ into a form, in which the lower and upper bounds of each element x_i of the vector $\vec{x}$ is expressed in terms of the elements $x_1, \ldots, x_{i-1}$ only. This fact is very important when using a nested loop, in order to traverse an iteration space J^n defined by a system of inequalities. In this case, the bounds of index j_k of the nested loop must be expressed in terms of the $k-1$ outer indexes only. This means that FME method can convert a system describing a general iteration space into a form suitable for use in nested loops.

After applying FME, the eliminated system consists of a very large number of inequalities describing the bounds of each variable x_i, but some of them are not necessary for the calculation of x_i's bounds. The unnecessary inequalities must be eliminated to simplify the resulting system. In order to remove the redundant inequalities, two methods have been proposed: the "Ad-Hoc simplification method" and the "Exact simplification method". A full description of the Fourier-Motzkin elimination method, the Ad-Hoc simplification and the Exact simplification is presented in [6].

If the initial system of inequalities consists of k inequalities with n variables, then the complexity of the FME algorithm can be expressed by the form: $Complexity = O(\frac{k^{2^n}}{2^{2(n+1)-2}}) \approx O((\frac{k}{2})^{2^n})$ [22]. FME is an extremely complex method, since it depends *doubly exponentially* on the number of loops involved.

3 Tiling (Supernode) Transformation

In a tiling transformation, the iteration space J^n is partitioned into identical n-dimensional parallelepiped areas (tiles or supernodes), formed by n independent families of parallel hyperplanes. Tiling transformation is defined by the n-dimensional square matrix H. Each row vector of H is perpendic-

ular to one family of hyperplanes forming the tiles. Dually, tiling transformation can be defined by n linearly independent vectors, which are the sides of the tiles. Similar to matrix H, matrix P contains the side-vectors of a tile as column vectors. It holds $P = H^{-1}$. Formally, tiling transformation is defined as follows:

$$r : Z^n \longrightarrow Z^{2n}, r(j) = \begin{bmatrix} \lfloor Hj \rfloor \\ j - H^{-1}\lfloor Hj \rfloor \end{bmatrix},$$

where $\lfloor Hj \rfloor$ identifies the coordinates of the tile that iteration point $j \in J^n$ is mapped to and $j - H^{-1}\lfloor Hj \rfloor$ gives the coordinates of j within that tile relative to the tile origin. Thus, the initial n-dimensional iteration space J^n is transformed to a $2n$-dimensional one, consisting of the n-dimensional space of tiles and the n-dimensional space of indexes within tiles. The following spaces are derived from a tiling transformation H, when applied to an iteration space J^n.

1. The Tile Iteration Space $TIS(H) = \{j \in Z^n | 0 \leq \lfloor Hj \rfloor < 1\}$, which contains all points that belong to the tile starting at the axes origins.

2. The Tile Space $J^S(J^n, H) = \{j^S | j^S = \lfloor Hj \rfloor, j \in J^n\}$, which contains the images of all points $j \in J^n$ according to tiling transformation.

3. The Tile Origin Space $TOS(J^S, H^{-1}) = \{j \in Z^n | j = H^{-1}j^S, j^S \in J^S\}$, which contains the origins of tiles in the original space.

Following the above, it holds: $J^n \xrightarrow{H} J^S$ and $J^S \xrightarrow{P} TOS$. For simplicity reasons, we will refer to $TIS(H)$ as TIS, $J^S(J^n, H)$ as J^S and $TOS(J^S, H^{-1})$ as TOS. Note that all points of J^n that belong to the same tile, are mapped to the same point of J^S. Note also that TOS is not necessarily a subset of J^n, since there may exist tile origins which do not belong to the original iteration space J^n, but some

iterations within these tiles do belong to J^n. The following example analyzes the properties of each of the spaces defined above.

Example 1 *Consider the following nested loop:*

```
FOR (j1 = 0; j1 <= 39)
    FOR (j2 = 0; j2 <= 29)
        A[j1, j2] = A[j1 - 1, j2 - 2] + A[j1 - 3, j2 - 1];
    ENDFOR
ENDFOR
```

The corresponding iteration space J^2 is: $J^2 = \{(j_1, j_2) | 0 \leq j_1 \leq 39, 0 \leq j_2 \leq 29\}$. *Let us apply a tiling transformation defined by matrix* $H = \begin{bmatrix} \frac{1}{5} & -\frac{1}{10} \\ -\frac{1}{20} & \frac{3}{20} \end{bmatrix}$ *or, equivalently, by* $P = \begin{bmatrix} 6 & 4 \\ 2 & 8 \end{bmatrix}$, *which is legal [28] (since $HD \geq 0$) and has both communication and scheduling-optimal shape ([7], [16], [17], [18], [39]), for the specific problem. Then, as shown in Figure 2a, TIS contains the points* $\{(0,0), (1,1), (1,2), (2,1), (2,2), (2,3), (2,4), \ldots, (7,5), (7,6), (7,7), (7,8), (8,7), (8,8), (8,9), (9,9)\}$. *In addition, as shown in Figure 2c, J^n is transformed by matrix H to the Tile Space* $J^S = \{(-3,3), (-3,4), (-2,1), (-2,2), (-2,3), (-2,4), \ldots, (6,-2), (6,-1), (6,0), (7,-2), (7,-1)\}$. *In the sequel, as shown by the grey dots in Figure 2b, the Tile Space J^S is transformed by matrix P to* $TOS = \{(-6,18), (-2,26), (-8,4), (-4,12), (0,20), (4,28), \ldots, (28,-4), (32,-4), (36,12), (34,-2), (38,6)\}$. □

Points belonging to the same tile with tile origin $j_0 \in TOS$, satisfy the system of inequalities $0 \leq H(j - j_0) < 1$. In order to deal with integer inequalities, we define g to be the smallest integer such that gH is an integer matrix. Thus, we can rewrite the above system of inequalities as follows: $0 \leq gH(j - j_0) < g \Leftrightarrow 0 \leq gH(j - j_0) \leq (g-1)$. We denote $S = \begin{pmatrix} gH \\ -gH \end{pmatrix}$ and $\vec{s} = \begin{pmatrix} (g-1)\vec{1} \\ \vec{0} \end{pmatrix}$. Equivalently, the above system becomes: $S(j - j_0) \leq \vec{s}$. Note that if $j_0 = 0$, $S(j - j_0) \leq \vec{s}$ is satisfied only by points in TIS.

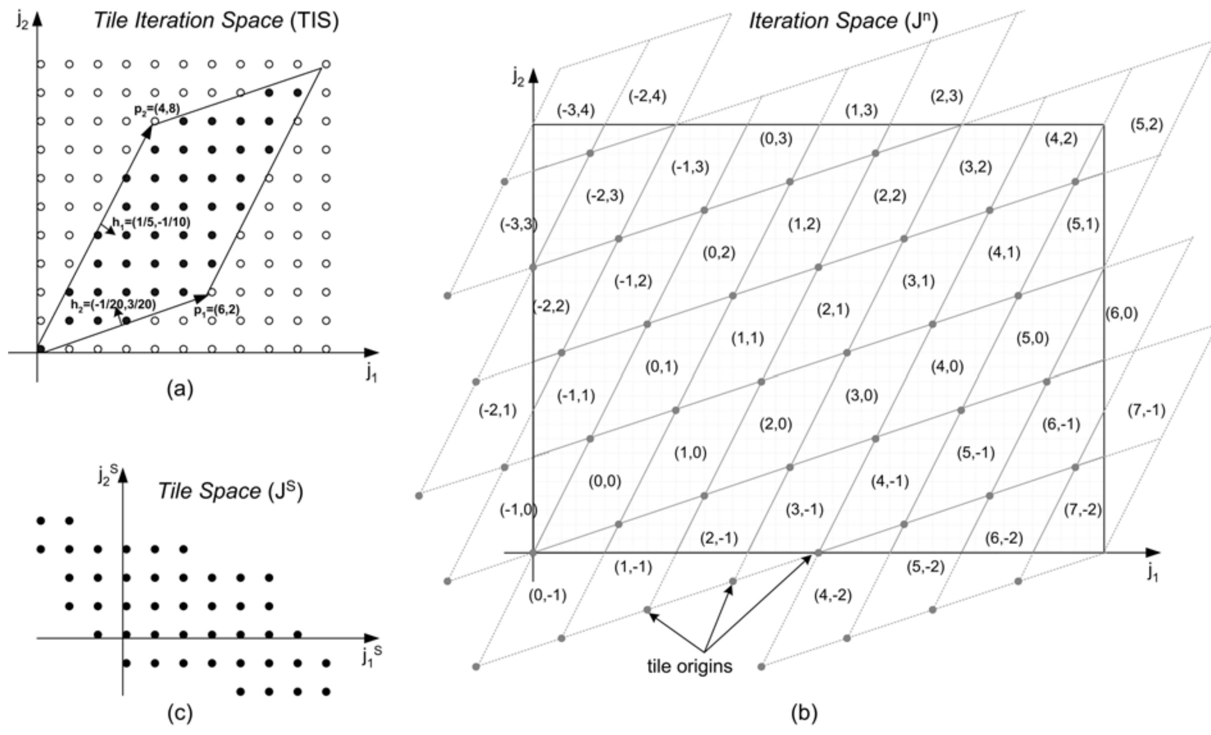

Figure 2. Example Spaces

Example 2 *In the loop of Example 1, the set of inequalities describing the iteration space J^2 is:*

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} j_1 \\ j_2 \end{pmatrix} \leq \begin{pmatrix} 39 \\ 29 \\ 0 \\ 0 \end{pmatrix}.$$

Using the same tiling transformation matrix $H = \begin{bmatrix} \frac{1}{5} & -\frac{1}{10} \\ -\frac{1}{20} & \frac{3}{20} \end{bmatrix}$, *the system of inequalities* $S(j - j_0) \leq \vec{s}$ *describing a tile is (since* $g = 20$*):*

$$\begin{pmatrix} 4 & -2 \\ -1 & 3 \\ -4 & 2 \\ 1 & -3 \end{pmatrix} \begin{pmatrix} j_1 - j_{0_1} \\ j_2 - j_{0_2} \end{pmatrix} \leq \begin{pmatrix} 19 \\ 19 \\ 0 \\ 0 \end{pmatrix}. \qquad \square$$

4 Code Generation Methods

In this section, we elaborate on generating tiled code that will traverse an iteration space J^n transformed by a tiling transformation. We call this code *sequential tiled code*. By applying tiling to J^n, we obtain the Tile Space J^S, the Tile Iteration Space TIS and the Tile Origin Space TOS. In Section 3, it was shown that tiling transformation is a $Z^n \longrightarrow Z^{2n}$ transformation, which means that a point $j \in J^n$ is transformed into a tuple of n-dimensional factors (j_a, j_b), where j_a identifies the tile that the original point belongs to ($j_a \in J^S$) and j_b identifies the coordinates of the point relevant to the tile origin ($j_b \in TIS$). The sequential tiled code reorders the execution of indexes enforced by the original nested loop, resulting in an execution order described by the following scheme: `FOR (EVERY tile IN Tile Space` J^S`) TRAVERSE THE POINTS IN ITS INTERIOR`. According to the above, the sequential tiled code consists of a $2n$-dimensional nested loop. The n outermost loops traverse the Tile Space J^S, using indexes $t_1^S, t_2^S, \ldots, t_n^S$, and the n innermost loops traverse the points within the tile defined by $t_1^S, t_2^S, \ldots, t_n^S$, using indexes $j_1', j_2', \ldots, j_n'$. We denote l_k^S, u_k^S the lower and upper bounds of index t_k^S, respectively. Similarly, we denote l_k', u_k' the lower and upper bounds of index j_k'. In all cases, lower bounds L_k are of the form: $L_k = max(l_{k,0}, l_{k,1}, \ldots)$ and upper bounds U_k of the form: $U_k = min(u_{k,0}, u_{k,1}, \ldots)$, where $l_{k,j}$, $u_{k,j}$ are affine functions of the outermost indexes. By calculating factors $l_1^S, \ldots, l_n^S$, $u_1^S, \ldots, u_n^S$, $l_1', \ldots, l_n'$ and $u_1', \ldots, u_n'$, we can traverse the tiled iteration space as described before.

4.1 Previous Work

The problem of generating sequential tiled code can be separated into two subproblems: traversing the Tile Space J^S and sweeping the internal points of every tile or, in our context, finding lower and

upper bounds for the n outermost indexes $t_1^S, t_2^S, \ldots, t_n^S$ and finding lower and upper bounds for the n innermost indexes $j_1', j_2', \ldots, j_n'$. Ancourt and Irigoin in [4] dealt with these subproblems, by constructing an appropriate set of inequalities for each case. In order to traverse the Tile Space J^S, the first system is constructed by merging the inequalities representing the original iteration space and the inequalities representing a tile. Recall from Section 3 that a point $j \in J^n$ belonging to a tile with tile origin $j_0 \in TOS$, satisfies the set of inequalities: $S(j - j_0) \leq \vec{s}$. Since $j_0 = Pj^S$, the preceding system of inequalities becomes: $\begin{pmatrix} -gI & gH \\ gI & -gH \end{pmatrix} \begin{pmatrix} j^S \\ j \end{pmatrix} \leq \vec{s}$. Recall also that a point $j \in J^n$ satisfies the system of inequalities $Bj \leq \vec{b}$. Combining these systems, we obtain the final system of inequalities:

$$\begin{pmatrix} 0 & B \\ -gI & gH \\ gI & -gH \end{pmatrix} \begin{pmatrix} j^S \\ j \end{pmatrix} \leq \begin{pmatrix} \vec{b} \\ \vec{s} \end{pmatrix} \tag{1}$$

In order to traverse the internal points of every tile, the above set of inequalities is rewritten equivalently:

$$\begin{pmatrix} B \\ gH \\ -gH \end{pmatrix} j \leq \begin{pmatrix} \vec{b} \\ (g-1)\vec{1} + gj^S \\ \vec{0} - gj^S \end{pmatrix}, \tag{2}$$

where vector j^S gives the coordinates of the tile to be traversed. Ancourt and Irigoin propose the application of FME method to the above systems in order to obtain proper formulas for the lower and upper bounds of the $2n$-dimensional loop that will traverse the tiled space.

4.2 Our Method

We will now introduce an alternative method to generate tiled code. The concept of dividing the main problem into the subproblems of traversing the Tile Space J^S and sweeping the internal points of every tile, is preserved here as well. However, we refine complexity by applying certain transformations to both subproblems, before constructing the final sets of inequalities and applying FME method to them. That is, we reduce the inequalities involved in the derived systems and, consequently, reduce the FME method steps (compilation time reduction) and the number of expressions required to calculate the loop bounds (run time reduction).

4.2.1 Enumerating the Tiles

The subproblem of traversing the Tile Space J^S has been considered by many authors as an example of applying the non-unimodular tiling transformation to the original iteration space. More specifically, Ramanujam in [26] and [27] applied the non-unimodular tiling transformation to the set of inequalities $Bj \leq \vec{b}$ describing the iteration space, as follows: $Bj \leq \vec{b} \Rightarrow BH^{-1}Hj \leq \vec{b} \Rightarrow$

$$BPj^S \leq \vec{b} \tag{3}$$

Here again, the application of FME method to the derived system of inequalities is proposed, in order to obtain closed form formulas for tile bounds $l_1^S, \ldots, l_n^S$ and $u_1^S, \ldots, u_n^S$.

Unfortunately, the previous approach fails to enumerate tiles exactly. This is because the system of inequalities in (3) is satisfied by points in the Tile Space J^S, whose tile origins belong to J^n. However, as stated in Section 3, there exist some points in TOS that do not belong to J^n. Although these points do

not satisfy the preceding systems of inequalities, they must be traversed as well. In Figure 2b, tiles in the lower boundaries, such as (-3,3), (-2,1), (4,-2) and others, are not scanned by this method, because their origins do not belong to the original iteration space J^n. Consequently, a modification is required, so that FME method can scan all tiles correctly. As shown in Figure 4, what is needed is a proper reduction of the lower bounds and/or a proper increase of the upper bounds of our space, in order to include all tile origins. Lemma 1 determines how much we must expand space bounds, in order to include all points of TOS.

Lemma 1 *If we apply tiling transformation P to an iteration space J^n, whose bounds are expressed by the system of inequalities $Bj \leq \vec{b}$, then for all tile origins $j_0 \in TOS$, it holds:*

$$Bj_0 \leq \vec{b'}, \tag{4}$$

where $\vec{b'}$ is determined by the expression:

$$b'_i = b_i + \frac{g-1}{g} \sum_{r=1}^{n} (\vec{\beta_i} \vec{p_r})^-, i = 1, \ldots, n \tag{5}$$

where $\vec{\beta_i}$ is the i-th row of matrix B, $\vec{p_r}$ is the r-th column of matrix P and $(\vec{\beta_i}\vec{p_r})^- = max(-\vec{\beta_i}\vec{p_r}, 0)$.

Proof: *Given in Appendix*

If we work with the Tile Space J^S and take into account that $j_0 = Pj^S$, we equivalently get the system of inequalities:

$$BPj^S \leq \vec{b'} \tag{6}$$

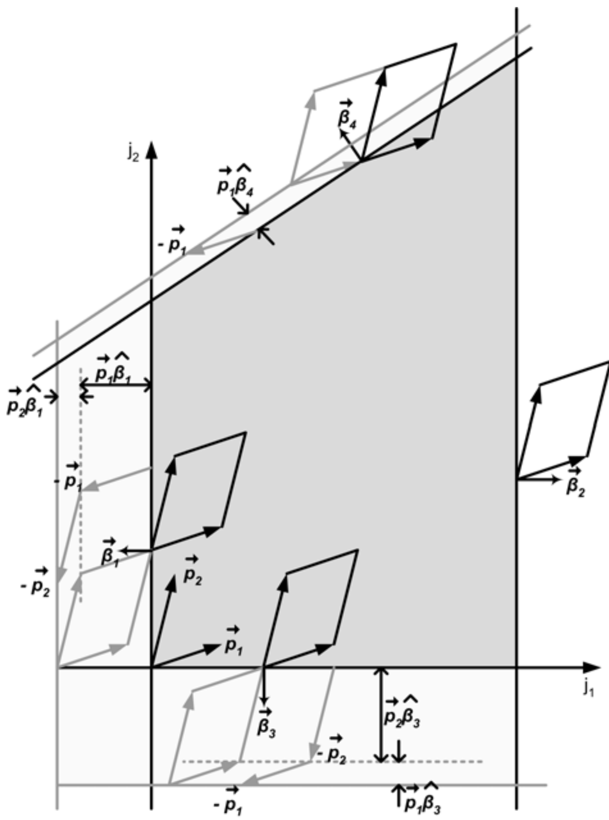

Figure 3. Expanding bounds to include all tile origins

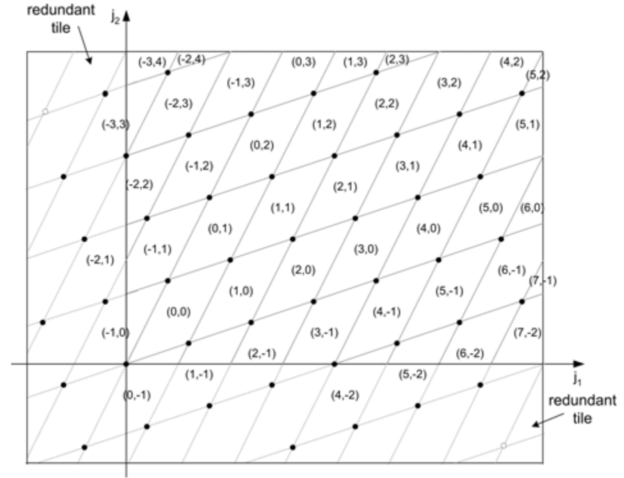

Figure 4. Example 3: expansion of bounds

Geometrically, the term added to each element of $\vec{b}$ expresses a parallel shift of the corresponding bound of the initial space. In Figure 3, we present an example of our method. Each row $\vec{\beta_i}$ of matrix B expresses a vector vertical to the corresponding bound of the iteration space with its direction outwards. The equation of this boundary surface is $\vec{\beta_i}\vec{x} = b_i$. A parallel shift of this surface by a vector $\vec{x_0}$ is expressed by the equation $\vec{\beta_i}(\vec{x} - \vec{x_0}) = b_i \Leftrightarrow \vec{\beta_i}\vec{x} = b_i + \vec{\beta_i}\vec{x_0}$. As shown in Figure 3, we shift a boundary surface by vector $-\vec{p_r}$, iff the tile edge-vector $\vec{p_r}$ forms an angle greater than 90° with vector $\vec{\beta_i}$ (as the angles between the vectors $\vec{\beta_1}$ and $\vec{p_1}$, $\vec{\beta_1}$ and $\vec{p_2}$, $\vec{\beta_3}$ and $\vec{p_1}$, $\vec{\beta_3}$ and $\vec{p_2}$, $\vec{\beta_4}$ and $\vec{p_1}$ of Figure 3), or, equivalently, iff $\vec{p_r}\vec{\beta_i} < 0$. This fact can be expressed as follows: if the dot product of $\vec{p_r}$, one of the columns of the matrix P, and $\vec{\beta_i}$, a row of B, is negative, then we subtract this dot product from the

constant b_i. Equivalently, in formula (5) we add the term $(\vec{\beta_i}\vec{p_r})^-$ to the constant b_i for all vectors $\vec{p_r}$. The multiplying factor $\frac{g-1}{g}$ expresses the fact that a tile is a semiopen hyperparallelepiped and thus we need not contain in the tile space the tiles which just touch the initial iteration space. Note, however, that this expansion of bounds may include some redundant tiles, whose origins belong to the extended space, but their internal points remain outside the original iteration space. These tiles will be accessed, but their internal points will not be swept, as it will be shown next, thus imposing little computation overhead in the execution of the sequential tiled code.

Example 3 *We will enumerate the tiles generated by the tiling transformation described in Examples 1 and 2. Following our approach, we should construct the system of inequalities in (6) making use of the expression in (5). Expression (5) in our case gives* $\vec{b'} = \begin{bmatrix} 39 & 29 & 9.5 & 9.5 \end{bmatrix}^T$ *and thus, the system in (6) becomes:*

$$\begin{pmatrix} 6 & 4 \\ 2 & 8 \\ -6 & -4 \\ -2 & -8 \end{pmatrix} \begin{pmatrix} j_1^S \\ j_2^S \end{pmatrix} \leq \begin{pmatrix} 39 \\ 29 \\ 9.5 \\ 9.5 \end{pmatrix}$$

. The expansion of bounds for this example is shown in Figure 4. A rough application of FME multiplies row 1 by 2 and adds it to row 4. Thus, we get $10j_1^S \leq 87.5 \Rightarrow j_1^S \leq 8$. *Similarly, we get* $j_1^S \geq -4$. *Consequently, a loop that enumerates the origins of tiles in our case has the form:*

```
FOR (j1^S = -4; j1^S <= 8)
   FOR (j2^S = max(⌈(-9.5-6j1^S)/4⌉, ⌈(-9.5-2j1^S)/8⌉); j2^S <= min(⌊(39-6j1^S)/4⌋, ⌊(29-2j1^S)/8⌋))
      ...
   ENDFOR
ENDFOR
```

Note that tiles $(8, -3)$ *and* $(-4, 4)$ *are redundant (Fig. 4).* □

4.2.2 Scanning the points within a tile

As far as scanning the internal points of a tile is concerned, we present a new method based on the use of a non-unimodular tranformation. Our goal is to traverse the TIS and then slide the points of TIS properly, so as to scan all points of J^n. In order to achieve this, we transform the TIS to a rectangular space, called the Transformed Tile Iteration Space ($TTIS$). We traverse the $TTIS$ with an n-dimensional nested loop and then transform the indexes of the loop so as to return to the proper points of the TIS. In other words, we are searching for a transformation pair (P', H'): $TTIS \xrightarrow{P'} TIS$ and $TIS \xrightarrow{H'} TTIS$ (Fig. 5). Intuitively, we demand P' to be parallel to tile sides, that is, the column vectors of P' to be parallel to the column vectors of P. This is equivalent to the row vectors of H' being parallel to the row vectors of H. In addition to this, we demand the lattice of H' to be an integer space for loop indexes to be able to traverse it. Formally, we must find an n-dimensional transformation $H' : H' = VH$, where V is an $n \times n$ diagonal matrix and $\mathcal{L}(H') \subseteq Z^n$. The following Lemma proves that the second requirement is satisfied if and only if H' is integral.

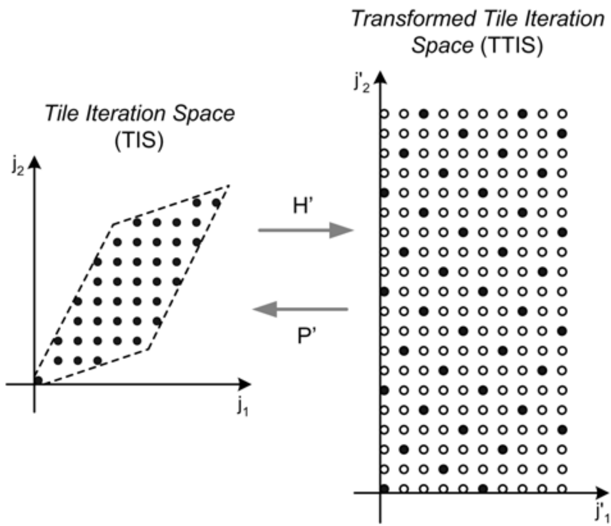

Figure 5. Traverse the TIS with a non-unimodular transformation

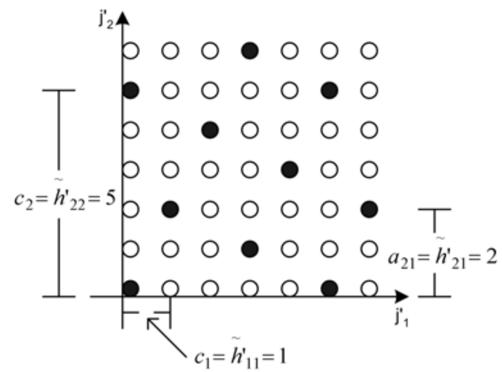

Figure 6. Steps and initial offsets in $TTIS$ derived from matrix $\widetilde{H'}$

Lemma 2 $j' = Aj \in Z^n\ \forall j \in Z^n$ *iff A is integral.*

Proof: *Given in Appendix*

Let us construct V in the following way: Every diagonal element v_{kk} is the smallest integer such that $v_{kk}h_k$ is integral, where h_k is the k-th row of matrix H. Thus, both requirements for H' are satisfied. It is obvious that H' is a non-unimodular transformation. This means that the Transformed Tile Iteration Space contains holes. In Figure 5, the holes in the $TTIS$ are depicted with white dots, while the actual points are depicted with black ones. So, in order to traverse the TIS, we have to scan all actual points of the $TTIS$ and then transform them back using matrix P'. We can apply any of the methods presented in [26], [27], [38], [25], [12] to traverse the $TTIS$. However, we will avoid the application of FME method by taking advantage of the tile shape regularity.

We use an n-dimensional nested loop with iterations indexed by $j'(j'_1, j'_2, \ldots, j'_n)$, in order to traverse the actual points of the $TTIS$. The bounds of the indexes j'_k are easily determined: it holds $0 \leq j'_k \leq v_{kk} - 1$. However, the increment step c_k of an index j'_k is not necessarily 1. In addition to this, if index j'_k is incremented by c_k, all indexes $j'_{k+1}, \ldots, j'_n$ should be initialized at certain offset values $a_{(k+1)k}, \ldots, a_{nk}$. Suppose that for a certain index vector j', it holds $P'j' \in Z^n$. The first question is how much to increment the innermost index j'_n so that the next swept point is also integral. Formally, we search the minimum $c_n \in Z$ such that $P' \begin{bmatrix} j'_1 & j'_2 & \ldots & j'_n + c_n \end{bmatrix}^T \in Z^n$. After determining c_n, the next step is to calculate the increment step of index j'_{n-1} so that the next swept point is also integral. In this case, it is possible that index j'_n should also be incremented by an offset $a_{n(n-1)}$: $0 \leq a_{n(n-1)} < c_n$. In the general case of index j'_k we need to determine $c_k, a_{(k+1)k}, \ldots, a_{nk}$ such that: $P' \begin{bmatrix} j'_1 & \ldots & j'_k + c_k & j'_{k+1} + a_{(k+1)k} & \ldots & j'_n + a_{nk} \end{bmatrix}^T \in Z^n$. Every index j'_k has $k-1$ different incremental offsets a_{ki}, depending on each of the increment steps c_i of the $k-1$ outer indexes j'_i.

These offsets are $a_{k1}, \ldots, a_{k(k-1)}$. The following Lemma proves that increment steps c_k and offsets a_{kl}, ($k = 1 \ldots n$ and $l = 1 \ldots k-1$), are directly obtained from the Hermite Normal Form of matrix H', denoted $\widetilde{H'}$.

Lemma 3 *If $\widetilde{H'}$ is the column HNF of H' and $j'(j'_1, j'_2, \ldots, j'_n)$ is the index vector used to traverse the actual points of $\mathcal{L}(H')$, then the increment step (stride) for index j'_k is $c_k = \widetilde{h'}_{kk}$ and the incremental offsets are $a_{kl} = \widetilde{h'}_{kl}$, ($k = 1 \ldots n$ and $l = 1 \ldots k-1$).*

Proof: *Given in Appendix*

According to the above analysis, the point that will be traversed using the next instantiation of indexes is calculated from the current instantiation, since steps and incremental offsets are added to the current indexes. Special care is taken so that every time the index vector $j' = (j'_1, \ldots, j'_n)$ is to be modified, the new index vector j' is calculated as a sum of current j' and a multiple of a column-vector of $\widetilde{H'}$. Thus, assuming that the current instantiation $j' \in \mathcal{L}(H')$, we ensure that the next point to be traversed remains in $\mathcal{L}(H')$.

Theorem 3 *The following n-dimensional nested loop traverses all points $j' \in TTIS$*

FOR ($j'_1 = 0, \ldots, j'_n = 0$; $j'_1 \leq v_{11} - 1$; $j'_1 += \widetilde{h'}_{11}, \ldots, j'_n += \widetilde{h'}_{n1}$)
  FOR ($j'_n += \lceil \frac{-j'_2}{\widetilde{h'}_{22}} \rceil * \widetilde{h'}_{n2}, \ldots, j'_2 += \lceil \frac{-j'_2}{\widetilde{h'}_{22}} \rceil * \widetilde{h'}_{22}$; $j'_2 \leq v_{22} - 1$; $j'_2 += \widetilde{h'}_{22}, \ldots, j'_n += \widetilde{h'}_{n2}$)
    ...
    FOR ($j'_n += \lceil \frac{-j'_n}{\widetilde{h'}_{nn}} \rceil * \widetilde{h'}_{nn}$; $j'_n \leq v_{nn} - 1$; $j'_n += \widetilde{h'}_{nn}$)
      ...
    ENDFOR
    ...
  ENDFOR
ENDFOR

Proof: *It can be easily derived from Lemmas 2 and 3.*

We now need to adjust the above loop, which sweeps all points in $TTIS$, in order to traverse the internal points of any tile in J^S. If $j' \in TTIS$ is the point that is derived from the indexes of the

former loop and $j^S \in J^S$ is the tile, whose internal points $j \in J^n$ we want to traverse, it will hold: $j = Pj^S + P'j' = j_0 + P'j'$, $j_0 \in TOS$, where $j_0 = Pj^S$ is the tile origin, and $P'j' \in TIS$ is the corresponding to j' point in TIS. Special attention also needs to be paid so that the points traversed do not overcome the original space boundaries. As we have mentioned before, a point $j \in J^n$ satisfies the following set of inequalities: $Bj \leq \vec{b}$. Replacing j by the above equation we have: $B(j_0 + P'j') \leq \vec{b} \Rightarrow$

$$BP'j' \leq \vec{b} - Bj_0 \tag{7}$$

By applying FME method to the preceding set of inequalities, we obtain proper expressions for j', so that we do not cross the original space boundaries. In this way, the problem of redundant tiles that arose in the previous section is also faced, since no computation is performed in these tiles.

Example 4 *Let us consider the same algorithm as in the previous examples. We will now sweep the internal points of a tile. If we follow our method, we have the following:* $H' = \begin{bmatrix} 2 & -1 \\ -1 & 3 \end{bmatrix}$ *and* $V = \begin{bmatrix} 10 & 0 \\ 0 & 20 \end{bmatrix}$. *Accordingly,* $P' = \begin{bmatrix} \frac{3}{5} & \frac{1}{5} \\ \frac{1}{5} & \frac{2}{5} \end{bmatrix}$. *The Hermite Normal Form of matrix* H' *is* $\widetilde{H'} = \begin{bmatrix} 1 & 0 \\ 2 & 5 \end{bmatrix} = \begin{bmatrix} 2 & -1 \\ -1 & 3 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix}$ *and thus, as shown in Figure 6,* $c_1 = \widetilde{h'}_{11} = 1$, $c_2 = \widetilde{h'}_{22} = 5$, $a_{21} = \widetilde{h'}_{21} = 2$. *Consequently, the code that traverses the indexes inside every internal tile, according to Theorem 3, is:*

$\begin{pmatrix} j_{01} \\ j_{02} \end{pmatrix} = \begin{bmatrix} 6 & 4 \\ 2 & 8 \end{bmatrix} \begin{pmatrix} j_1^S \\ j_2^S \end{pmatrix}$;

FOR $(j_1' = 0, j_2' = 0;\ j_1' \leq 9;\ j_1' += 1, j_2' += 2)$

  FOR $(j_2' += \lceil \frac{-j_2'}{5} \rceil * 5;\ j_2' \leq 19;\ j_2' += 5)$

    $\begin{pmatrix} j_1 \\ j_2 \end{pmatrix} = \begin{pmatrix} j_{0_1} \\ j_{0_2} \end{pmatrix} + \begin{bmatrix} \frac{3}{5} & \frac{1}{5} \\ \frac{1}{5} & \frac{2}{5} \end{bmatrix} \begin{pmatrix} j_1' \\ j_2' \end{pmatrix}$;

    $A[j_1, j_2] = A[j_1 - 1, j_2 - 2] + A[j_1 - 3, j_2 - 1]$;

  ENDFOR

ENDFOR

In order to exactly scan the internal of boundary tiles, we construct matrix $[BP'|\vec{b}|B] =$

$$\begin{pmatrix} \frac{3}{5} & \frac{1}{5} & 39 & 1 & 0 \\ \frac{1}{5} & \frac{2}{5} & 29 & 0 & 1 \\ -\frac{3}{5} & -\frac{1}{5} & 0 & -1 & 0 \\ -\frac{1}{5} & -\frac{2}{5} & 0 & 0 & -1 \end{pmatrix}$$

. *FME method on this matrix gives:*

$$\begin{pmatrix} 1 & 0 & 78 & 2 & -1 \\ \frac{3}{5} & \frac{1}{5} & 39 & 1 & 0 \\ \frac{1}{5} & \frac{2}{5} & 29 & 0 & 1 \\ -1 & 0 & 29 & -2 & 1 \\ -\frac{3}{5} & -\frac{1}{5} & 0 & -1 & 0 \\ -\frac{1}{5} & -\frac{2}{5} & 0 & 0 & -1 \end{pmatrix}$$

. *Consequently, the code that traverses the indexes inside tiles, which cut the iteration space bounds, is:*

```
(j01, j02)^T = [6 4; 2 8] (j1^S, j2^S)^T;
lb1 =max(0, -29 - 2j01 + j02);
ub1 =min(9 /* v11-1 */, 78 - 2j01 + j02);
FOR (j1' = lb1, j2' = lb1 * 2; j1' <= ub1; j1'+ = 1, j2'+ = 2)
  lb2 =max(0, -3j1' - 5j01, ceil((-j1' - 5j02)/2));
  ub2 =min(19 /* v22-1 */, -3j1' - 5j01 + 195, floor((-j1' - 5j02 + 145)/2));
  FOR (j2'+ = ceil((lb2 - j2')/5) * 5; j2' <= ub2; j2'+ = 5)
      (j1, j2)^T = (j01, j02)^T + [3/5 1/5; 1/5 2/5] (j1', j2')^T;
      A[j1, j2] = A[j1 - 1, j2 - 2] + A[j1 - 3, j2 - 1];
  ENDFOR
ENDFOR
```

□

5 Parallelization Aspects

In this section, we briefly refer to some parallelization aspects of the sequential tiled code. Recall that the parallelization of an arbitrarily tiled algorithm involves two separate tasks: first, the generation of the sequential tiled code and, second, the parallelization of this code. This paper focuses on the first task. Parallelization, in the sequel, can be separated in sub-tasks such as iteration distribution, data distribution and message-passing code generation. Our approach of accessing the internal of the tiles with the Hermite Normal Form of matrix H' can also provide significant benefits to the parallelization process, such as better data placement and thus, more efficient memory usage and ease of array data

addressing schemes.

When executing an algorithm on a distributed memory machine, the original data space of the algorithm is distributed to the local memories of the processing nodes. The local data space of each node is in general a non-rectangular subset of the original data space, even if rectangular tiling is applied [2]. However, applying the proposed transformations, each processor can iterate over a rectangular local iteration space (TTIS) and access rectangular data spaces as well. In this way, each processor can allocate exactly the required amount of memory. Rectangular data spaces also allow for straightforward addressing schemes of array elements and thus a direct way of sweeping data by the generated code. Another very important benefit in parallelization is the convenient determination of the communication sets. Each communication set contains the communication points, i.e. the points that are written in the local memory of a processor and are needed by another. The communication points have the following property: if we add one dependence vector to them, then the resulting point lies in the exterior of the tile. Figure 7 shows the communication points and sets when determined in the TIS and in the TTIS. d_1 and d_2 are the dependences of the original algorithm, while d'_1 and d'_2 are the transformed dependences in the $TTIS$. It is obvious that, when working with the rectangular $TTIS$, the communication sets are much more easily determined since they are rectangular as well. Details about the parallelization process only (iteration distribution, data distribution, message-passing code generation etc.) can be found in [14].

6 Comparison – Experimental Results

We have implemented both our method (in the sequel denoted as RI - Reduced Inequalities) and the one described in [4] by Ancourt and Irigoin (denoted as AI), as a software tool which automatically generates tiled C code using any tiling transformation P. In this section, we compare AI and RI methods

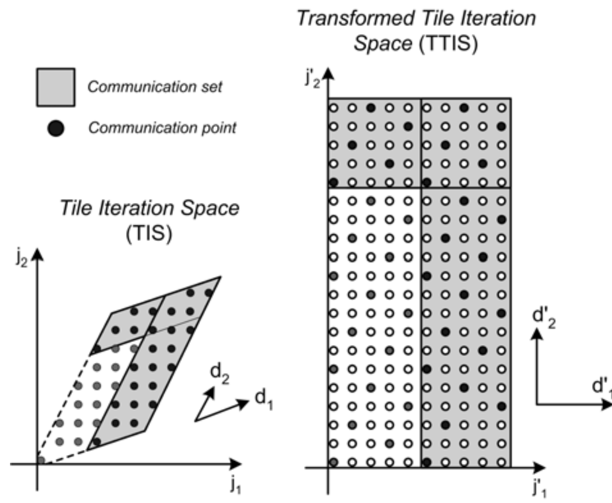

Figure 7. Determining Communication Sets in the TIS and TTIS

both in terms of compilation time and generated code efficiency. We generated several random $2-D$ and $3-D$ problems and measured the following: compilation time, row operations performed by FME and run time of the generated code. In the sequel, we applied both AI and RI methods to three real applications: SOR, Jacobi and ADI integration. We also applied the inequalities of AI method to the Omega calculator [23] and generated code for all problems. We then measured the compilation time and run time obtained by Omega (the results are denoted as AI-Omega) and compared them with the ones obtained by AI (using our tool) and RI. Table 1 shows the iteration spaces used as examples in $2-D$ and $3-D$ problems. We applied several tiling transformations, in which the non-zero elements of the tiling matrices were randomly generated. In $2-D$ spaces we applied three different tiling transformations (P_1, P_2, P_3) varying from the diagonal matrix P_1 to more complex ones. In $3-D$ spaces we applied seven different tiling transformations $(P_4, \ldots, P_{10})$, again here starting from the diagonal P_4 and adding non-zero elements (P_{10} contains no zero element). We performed our experiments on a PIII @ 800MHz processor with 128MB of RAM. The operating system is Linux with kernel 2.4.18. The generated tiled code was compiled using gcc v.2.95.4 with the -O3 optimization flag. We also experimented with

lower optimization levels, where the execution times were slower, but the relative results for all methods remained the same.

Table 1. Example Iteration Spaces

	i_1		i_2		i_3		
	lower bound	**upper bound**	**lower bound**	**upper bound**	**lower bound**	**upper bound**	**# of iterations**
Space1	-1999	4999	-1999	4999	-	-	48986001
Space2	-1999	4999	-1999	$4999 + 2i_1$	-	-	69983001
Space3	-4999	4999	$-4999 + 3i_1$	$4999 + 2i_1$	-	-	99980001
Space4	0	399	0	399	0	399	64000000
Space5	0	399	0	$399 + i_1$	0	399	95920000
Space6	0	399	$-i_1$	$399 + i_1$	0	399	127840000
Space7	-99	149	$-99 - i_1$	$149 + i_1$	-99	$149 + 2i_2$	22904099
Space8	0	399	$-i_1$	$399 + i_1$	i_1	$79 + 2i_2$	117635018
Space9	-99	149	$-99 - i_1$	$149 + i_1$	$-99 - i_1$	$149 + i_1 + 2i_2$	31129399
Space10	0	59	$-i_1$	$59 + i_1$	$-i_1 - 3i_2$	$59 + i_1 + 2i_2$	1994462

6.1 Row Operations - Compilation Time

Tables 2,3 summarize the results (row operations and compilation time) from the compilations of all iteration spaces tiled with all candidate tiling matrices. We present here the number of row operations and compilation times of one matrix (P_2 for $2 - Ds$ and P_7 for $3 - Ds$) for all iteration spaces and the average values of each matrix for all iteration spaces.

6.2 Run Time

In order to evaluate the run time overhead due to tiling, we executed all tiled codes of the previous problems and measured their run time. We also executed the original untiled serial code for each problem. We define the Tiling Overhead Factor - TOF as the fraction of the run time of the sequential tiled code to the run time of the untiled code: $TOF = \frac{\text{Run time of Sequential Tiled Code}}{\text{Run time of Untiled Code}}$. Note that, the loop body in each case is a simple array assignment statement and, thus, the run time measured is dominated by the

Table 2. FME Row Operations and Compilation Time (ms) for 2D Algorithms

	AI	RI	AI-Omega	AI	RI
Row Operations (P_2)			***Compilation Time*** (P_2)		
Space1	37	10	22.56	0.28	0.27
Space2	33	10	21.56	0.28	0.27
Space3	34	10	22.78	0.29	0.26
Avg. Row Operations			***Avg. Compilation Time***		
P_1	31	10	18.88	0.27	0.26
P_2	35	10	22.30	0.28	0.27
P_3	55	12	37.63	0.36	0.3

Table 3. FME Row Operations and Compilation Time (ms) for 3D Algorithms

	AI	RI	AI-Omega	AI	RI
Row Operations (P_7)			***Compilation Time*** (P_7)		
Space4	264	28	235.55	1.33	0.49
Space5	578	34	367.78	6.0	0.52
Space6	508	42	1,188.72	4.24	0.55
Space7	1411	38	911.38	40.78	0.54
Space8	1522	42	2,099.32	51.31	0.56
Space9	379	38	370.47	2.61	0.55
Space10	419	42	527.3	3.08	0.56
Avg. Row Operations			***Avg. Compilation Time***		
P_4	88	22	51.87	0.51	0.43
P_5	105	22	67.2	0.58	0.44
P_6	265	38	276.14	1.67	0.53
P_7	726	38	814.36	15.62	0.54
P_8	5382	36	2,901.14	1,746.64	0.53
P_9	3767	35	2,921.1	781.04	0.53
P_{10}	59563	41	2,531.58	356,508.91	0.56

time to compute the loop bounds. Since the array size was small (20×20) and the tile sizes were not chosen to be optimal for cache locality, the sequential tiled code does not present any improvement due to the exploitation of the memory hierarchy. Thus, TOF indicates the overhead imposed by the evaluation of the new loop bounds, due to tiling. If TOF is too large, it will aggravate the speedup obtained when we parallelize nested `FOR`-loops using tiling. Table 4 summarizes the tiling overhead factors. Again here we present the TOFs of P_2 and P_7 applied to all iteration spaces and the average TOFs of all Ps across all iteration spaces. Figure 8 shows the TOF of $3 - D$ problems as a function of the number of non-zero elements in tiling matrix P.

6.3 Real Applications

In our last set of experiments, we applied AI and RI methods to tile three real applications: SOR, Jacobi, and ADI integration. For the first two problems, there is a skewed and an unskewed version, and

Table 4. Tiling Overhead Factors (TOF) for $2-D$ and $3-D$ Problems

	AI-Omega	AI	RI		AI-Omega	AI	RI
TOF (2D) (P_2)				***Avg. TOF (2D)***			
Space1	6.27	4.55	1.61	P_1	2.85	1.03	1.31
Space2	6.12	4.62	1.63	P_2	6.62	4.78	1.69
Space3	7.45	5.16	1.82	P_3	8.23	6.41	3.75
TOF (3D) (P_7)				***Avg. TOF (3D)***			
Space4	15.50	9.86	4.65	P_4	1.99	1.26	1.17
Space5	16.09	10.05	5.14	P_5	4.96	3.44	1.88
Space6	16.20	10.10	5.29	P_6	9.55	7.16	4.62
Space7	12.67	9.04	4.80	P_7	13.90	9.47	5.17
Space8	12.72	8.92	4.65	P_8	11.24	9.14	3.60
Space9	11.80	8.95	4.84	P_9	10.74	8.78	5.51
Space10	12.29	9.38	6.84	P_{10}	13.62	11.07	5.62

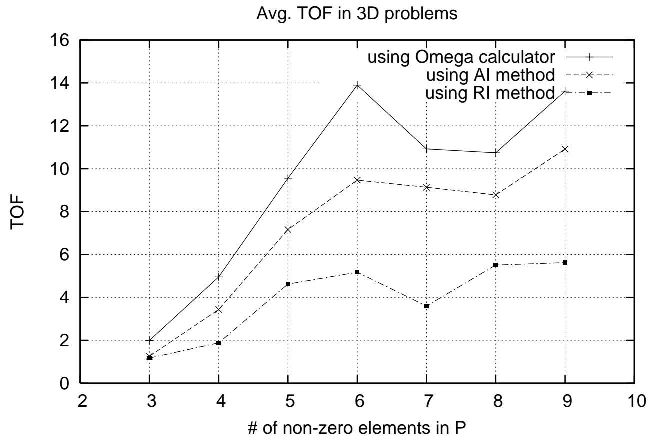

Figure 8. Avg. Tiling Overhead Factors for $3-D$ problems

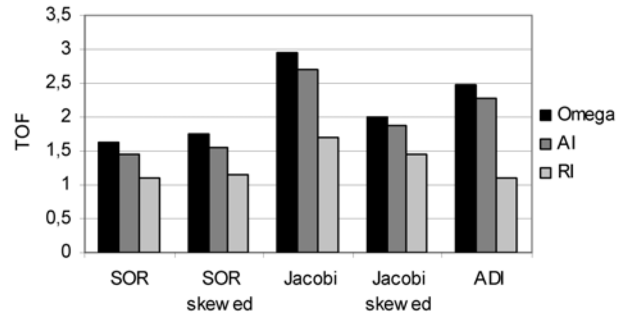

Figure 9. Tiling Overhead Factors for real applications

for each version there are four (communication and scheduling) optimal matrices as described in [17] and [39]. Table 5 summarizes the row operations, compilation times and TOFs for each case. Figure 9 shows the TOFs obtained by each method, in each case.

6.4 Overall Evaluation Comments

As far as compilation time is concerned, RI method clearly outperforms AI method. This is due to the fact that RI method feeds FME with the system in (6), which consists of $2n$ inequalities with n

Table 5. Performance for Real Applications

		Row Operations		*Compilation Time* (ms)			*TOF*		
		AI	**RI**	**AI-Omega**	**AI**	**RI**	**AI-Omega**	**AI**	**RI**
SOR	P_1	99	22	53.03	0.50	0.42	1.47	1.20	1.05
	P_2	107	22	50.27	0.53	0.42	1.50	1.21	1.01
	P_3	118	22	49.01	0.57	0.42	1.75	1.63	1.05
	P_4	165	40	90.04	0.77	0.5	1.80	1.78	1.30
SOR skewed	P_1	99	22	42.09	0.53	0.41	1.59	1.29	1.06
	P_2	107	22	40.60	0.53	0.42	1.60	1.29	1.06
	P_3	118	22	57.9	0.57	0.42	1.90	1.73	1.12
	P_4	165	40	91.97	0.77	0.51	1.95	1.86	1.34
Jacobi	P_1	645	28	346.99	5.3	0.46	2.08	1.91	1.57
	P_2	645	28	347.96	5.26	0.47	2.09	1.92	1.60
	P_3	800	28	362.5	8.86	0.47	2.06	1.90	1.56
	P_4	3207	46	1,353.55	194.88	0.53	5.58	5.09	2.10
Jacobi skewed	P_1	645	28	251.885	4.93	0.48	1.99	1.88	1.44
	P_2	645	28	248.27	4.98	0.47	1.98	1.87	1.46
	P_3	800	28	229.34	8.19	0.48	2.02	1.89	1.45
	P_4	691	28	238.82	5.95	0.47	2.01	1.88	1.43
ADI	P_1	180	28	47.42	0.85	0.46	1.46	1.47	1.07

variables, while AI method feeds FME with the system in (1), which consists of $4n$ inequalities with $2n$ variables. Recall that FME is a doubly exponential algorithm and thus the reduction in its input size imposed by our method causes significant reduction in the method's execution steps, as clearly seen by the number of row operations. Note also that the exact simplification method of FME was not applied in the presented experiments, since the gain in run time by the application of the method was inadequate to justify the vast increase in compilation times, especially in the case of AI method (3% average and 10% maximum gain in run time). In particular, while RI compilation times remained in the order of milliseconds when using exact simplification, AI compilation times increased dramatically (reached the order of an hour). This means that we can practically apply exact simplification to RI, in order to further improve the efficiency of the generated code.

Despite the reduction in compilation time imposed by RI, it seems that both AI and AI-Omega perform well in $2-D$ and $3-D$ problems (compilation times are less than one second). However, in problems of

larger dimensions, both AI and AI-Omega present several problems. We executed a number of randomly generated $4-D$ algorithms and observed that, at first, the compilation time of AI becomes impractical (several hours or even days). More importantly, AI failed to generate code for almost half of the problems due to lack of memory. Note that FME is also doubly exponential in space, so in several $4-D$ problems even 1GB of virtual memory was not sufficient to cover the needs of the method. On the other hand, AI-Omega also faced some problems with memory space (to a smaller extent than AI) but here again, in almost half of the problems, the system rose an overflow exception. Apparently, after a large number of row operations in $4-D$ algorithms, some coefficients exceeded the system's `MAXINT`. In all cases RI method succeeded in generating code, within some seconds in the worst case.

As far as run time is concerned, RI also exhibits a significant improvement in performance in all problems. In particular, as shown in Figure 8, as the number of non-zero elements in matrix P increases, the improvement of RI method becomes much more obvious. This means that RI method performs very well in complex problems where the tiling matrices contain many non-zero elements and the iteration spaces are non-rectangular. In addition, as shown in Figure 9, RI's performance is nearly optimal in simpler algorithms such as SOR, Jacobi and ADI, since the TOF in these cases is very close to one. Thus, RI performs very well in easy problems and sustains a remarkably good performance even when the tiling transformations and the shape of the iteration spaces become increasingly complex.

The improvement in the quality of the generated code caused by RI, is due to the fact that, although the code to enumerate the tiles is essentially similar in AI and RI, the code to traverse the internal points of the tiles is completely different. Our tool makes a distinction between boundary and internal tiles and generates different code to scan the internal points for both AI and RI (as in Example 4). In the case of boundary tiles, RI method results in fewer inequalities for the bounds of the Tile Space. Consequently,

fewer bound calculations are executed during run time. In the case of internal tiles, which are the vast majority in most problems, the code of RI consists of a loop with constant bounds ($0 \le j'_i \le v_{ii} - 1$ for $i = 1, \ldots, n$), while the code of AI includes a loop whose bounds are derived from the application of FME to the system $\begin{pmatrix} gH \\ -gH \end{pmatrix} (j - j_0) \le \begin{pmatrix} (g-1)\vec{1} \\ \vec{0} \end{pmatrix}$. It is clear that the calculation of loop bounds in the first case is much more efficient. Finally, note that the enumeration of some redundant tiles does not impose any significant overhead, since the number of redundant tiles is negligible. The same holds for the non-unimodular transformation used to access the internal points of the tiles. In this case, the additional operations due to the transformation are simple integer multiplications, while operations on extra variables are integer additions and assignment statements which are all efficiently executed by modern processors and optimized by any back-end compiler like gcc.

Summarizing, the compilation time reduction is due to the method used to enumerate the tiles of the Tile Space, while the run time reduction is mainly due to the transformation of a non-rectangular tile to a rectangular one.

7 Conclusions

In this paper, we proposed a novel approach for the problem of generating code for tiled nested loops. Our method is applied to general parallelepiped tiles and non-rectangular space boundaries as well. In order to generate code efficiently, we divided the original problem into the subproblems of enumerating the tiles and sweeping the points inside every tile. In the first case, we extended previous work on non-unimodular transformations in order to precisely traverse all tile origins. In the second case, we proposed the use of a non-unimodular transformation in order to transform the tile iteration space into a hyper-rectangle. Experimental results show that our method outperforms previous work in terms of both

compile and run times, since it constructs smaller systems of inequalities and traverses the points within non-rectangular tiles as if they were rectangular.

Appendix

Proof of Lemma 1: We suppose that the point $j \in J^n$ belongs to the tile with origin j_0. Then, j can be expressed as the sum of j_0 and a linear combination of the column-vectors of the tiling matrix P: $j = j_0 + \sum_{l=1}^{n} \lambda_l \vec{p_l}$. In addition, as referred in Section 3, the following equality holds: $\vec{0} \leq gH(j - j_0) \leq (g \vec{-} 1)$. The i-th row of this inequality can be rewritten as follows: $0 \leq \vec{h_i}(j - j_0) \leq \frac{g-1}{g}$, where $\vec{h_i}$ is the i-th row-vector of matrix $H = P^{-1}$. Therefore: $0 \leq \vec{h_i} \sum_{l=1}^{n} \lambda_l \vec{p_l} \leq \frac{g-1}{g}$. As $P = H^{-1}$ it holds that $\vec{h_i}\vec{p_i} = 1$ and $\vec{h_i}\vec{p_l} = 0$ if $i \neq l$. Consequently, the last form can be rewritten as follows: $0 \leq \lambda_i \leq \frac{g-1}{g}$ for all $i = 1, \ldots, n$.

For each $j \in J^n$ the system of inequalities $Bj = \vec{b}$ holds. The k-th row of this system can be written as follows: $\vec{\beta_k}\vec{j} \leq b_k$. We can rewrite the last inequality in terms of the corresponding tile origin as follows: $\vec{\beta_k}(\vec{j_0} + \sum_{i=1}^{n} \lambda_i \vec{p_i}) \leq b_k \Rightarrow \vec{\beta_k}\vec{j_0} \leq b_k - \vec{\beta_k}(\sum_{i=1}^{n} \lambda_i \vec{p_i})$

$$\Rightarrow \vec{\beta_k}\vec{j_0} \leq b_k - \sum_{i=1}^{n} \lambda_i (\vec{\beta_k}\vec{p_i}) \tag{8}$$

In addition, as we proved above, it holds $0 \leq \lambda_i \leq \frac{g-1}{g}$ for all $i = 1, \ldots, n$. If multiplied by $\vec{\beta_k}\vec{p_i}$ this inequality gives:

a) If $\vec{\beta_k p_i} \geq 0$: $\lambda_i \vec{\beta_k}\vec{p_i} \geq 0$

b) If $\vec{\beta_k}\vec{p_i} < 0$: $\lambda_i \vec{\beta_k}\vec{p_i} \geq \frac{g-1}{g}\vec{\beta_k}\vec{p_i}$

According to the definitions of the symbol $(\vec{\beta_k}\vec{p_i})^- = max(-\vec{\beta_k}\vec{p_i}, 0)$, the previous inequalities can in every case be rewritten as follows: $\lambda_i \vec{\beta_k}\vec{p_i} \geq -\frac{g-1}{g}(\vec{\beta_k}\vec{p_i})^- \Rightarrow -\lambda_i \vec{\beta_k}\vec{p_i} \leq \frac{g-1}{g}(\vec{\beta_k}\vec{p_i})^-$. If added for $i = 1, \ldots, n$, this inequality gives: $-\sum_{i=1}^{n} \lambda_i \vec{\beta_k}\vec{p_i} \leq \frac{g-1}{g} \sum_{i=1}^{n} (\vec{\beta_k}\vec{p_i})^-$.

Therefore, from the last form and the inequality (8), we conclude that $\vec{\beta}_k \vec{j}_0 \leq b_k + \frac{g-1}{g} \sum_{i=1}^{n} (\vec{\beta}_k \vec{p}_i)^-$. Thus, for each tile with origin j_0, which has at least one point in the initial iteration space, it holds that $B j_0 \leq \vec{b'}$, where the vector $\vec{b'}$ is constructed so as its k-th element is given by the form: $b'_k = b_k + \frac{g-1}{g} \sum_{i=1}^{n} (\vec{\beta}_k \vec{p}_i)^-$. $\square$

Proof of Lemma 2: If A is integral it is clear that $j' \in Z^n \forall j \in Z^n$. Suppose that $j' \in Z^n \forall j \in Z^n$. We shall prove that A is integral. Without lack of generality we select $j = \hat{u}_k$, where $\hat{u}_k$ is the k-th unitary vector, $\hat{u}_k = (u_{k1}, \ldots, u_{kn})$, $u_{kk} = 1, u_{kj} = 0, j \neq k$. Then, according to the above, $A\hat{u}_k = [\sum_{i=1}^{n} a_{1i} u_{k_i}, \sum_{i=1}^{n} a_{2i} u_{k_i}, \ldots, \sum_{i=1}^{n} a_{ni} u_{k_i}]^T = [a_{1k}, a_{2k}, \ldots, a_{nk}]^T \in Z^n$. This holds for all $\hat{u}_k, k = 1 \ldots n$, thus A is integral. $\square$

Proof of Lemma 3: It holds $\mathcal{L}(H') = \mathcal{L}(\widetilde{H'})$. Thus, $\vec{0} \in \mathcal{L}(H')$ and the columns of $\widetilde{H'}$ belong to $\mathcal{L}(H')$. Suppose $\vec{x} \in Z^n / \vec{0}$ with the following properties: $x_i = 0$ for $i < k$ and $0 \leq x_i \leq \widetilde{h'}_{ik}$ for $k \leq i \leq n$. It suffices to prove that $\vec{x} \notin \{\vec{0}, \vec{h_k}\} \Rightarrow \vec{x} \notin \mathcal{L}(H')$. Suppose that $\vec{x} \in \mathcal{L}(H')$, which means that $\exists j \in Z^n : \widetilde{H'} j = \vec{x}$. $\widetilde{H'}$ is a lower triangular non-negative matrix and thus it holds: $x_1 = \widetilde{h'}_{11} j_1 = 0 \Rightarrow j_1 = 0$. Similarly, $j_i = 0$ for $i < k$. In the sequel, it holds: $x_k = \widetilde{h'}_{kk} j_k$. According to the above, it holds: $0 \leq x_k = \widetilde{h'}_{kk} j_k \leq \widetilde{h'}_{kk} \Rightarrow 0 \leq j_k \leq 1$. In addition, $0 \leq x_{k+1} = \widetilde{h'}_{(k+1)k} j_k + \widetilde{h'}_{(k+1)(k+1)} j_{k+1} \leq \widetilde{h'}_{(k+1)k}$. Since $\widetilde{h'}_{(k+1)(k+1)} > \widetilde{h'}_{(k+1)k} \Rightarrow j_{k+1} = 0$. Similarly, $j_i = 0$ for $i > k+1$. Consequently either $\vec{x} = \vec{0}$, or $\vec{x}$ is the $k-th$ column of $\widetilde{H'}$, which is a contradiction. $\square$

Acknowledgments

We wish to express our profound gratitude to the anonymous reviewers for their suggestions, which considerably increased the clarity and quality of the original manuscript.

References

[1] V. Adve and J. Mellor-Crummey. Advanced Code Generation for High Performance Fortran. In *Languages, Compilation Techniques and Run Time Systems for Scalable Parallel Systems*, chapter 18, Lecture Notes in Computer Science Series. Springer-Verlag, 1997.

[2] A. Agarwal, D. Kranz, and V. Natarajan. Automatic Partitioning of Parallel Loops and Data Arrays for Distributed Shared-Memory Multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, 6(9):943–962, 1995.

[3] S. P. Amarasinghe and M. S. Lam. Communication Optimization and Code Generation for Distributed Memory Machines. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Albuquerque, NM, Jun 1993.

[4] C. Ancourt and F. Irigoin. Scanning Polyhedra with DO Loops. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP)*, pages 39–50, Williamsburg, VA, Apr 1991.

[5] R. Andonov, P. Calland, S. Niar, S. Rajopadhye, and N. Yanev. First Steps Towards Optimal Oblique Tile Sizing. In *8th International Workshop on Compilers for Parallel Computers*, pages 351–366, Aussois, Jan 2000.

[6] A.J.C. Bik and H.A.G. Wijshoff. Implementation of Fourier-Motzkin Elimination. In *First Annual Conference of the ASCI*, pages 377–386, The Netherlands, 1995.

[7] P. Boulet, A. Darte, T. Risset, and Y. Robert. (Pen)-ultimate tiling? *INTEGRATION, The VLSI Jounal*, 17:33–51, 1994.

[8] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. In *Proceedings of the Third Workshop on Compilers for Parallel Computers*, pages 121–160, Jul 1992.

[9] F. Desprez, J. Dongarra, and Y. Robert. Determining the Idle Time of a Tiling: New Results. *Journal of Information Science and Engineering*, 14:167–190, Mar 1997.

[10] E. D'Hollander. Partitioning and Labeling of Loops by Unimodular Transformations. *IEEE Trans. on Parallel and Distributed Systems*, 3(4):465–476, Jul 1992.

[11] I. Drossitis, G. Goumas, N. Koziris, G. Papakonstantinou, and P. Tsanakas. Evaluation of Loop Grouping Methods based on Orthogonal Projection Spaces. In *Proceedings of the International Conference on Parallel Processing*, pages 469–476, Toronto, Canada, Aug 2000.

[12] A. Fernandez, J. Llaberia, and M. Valero. Loop Transformations Using Nonunimodular Matrices. *IEEE Trans. on Parallel and Distributed Systems*, 6(8):832–840, Aug 1995.

[13] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran-D Language Specification. Technical Report TR-91-170, Dept. of Computer Science, Rice University, Dec 1991.

[14] G. Goumas, N. Drosinos, M. Athanasaki, and N. Koziris. Compiling Tiled Iteration Spaces for Clusters. In *Proceedings of the 2002 IEEE International Conference on Cluster Computing*, pages 360–369, Chicago, Illinois, Sep 2002.

[15] G. Goumas, A. Sotiropoulos, and N. Koziris. Minimizing Completion Time for Loop Tiling with Computation and Communication Overlapping. In *Proceedings of IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS'01)*, San Francisco, Apr 2001.

[16] E. Hodzic and W. Shang. On Supernode Transformation with Minimized Total Running Time. *IEEE Trans. on Parallel and Distributed Systems*, 9(5):417–428, May 1998.

[17] E. Hodzic and W. Shang. On Time Optimal Supernode Shape. *IEEE Trans. on Parallel and Distributed Systems*, 13(12):1220–1233, Dec 2002.

[18] K Hogstedt, L. Carter, and J. Ferrante. Determining the Idle Time of a Tiling. In *Principles of Programming Languages (POPL)*, pages 319–323, Jan 1997.

[19] K. Hogstedt, L. Carter, and J. Ferrante. Selecting Tile Shape for Minimal Execution time. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 201–211, 1999.

[20] K Hogstedt, L. Carter, and J. Ferrante. On the Parallel Execution Time of Tiled Loops. *IEEE Trans. on Parallel and Distributed Systems*, 14(3):307–321, Mar 2003.

[21] F. Irigoin and R. Triolet. Supernode Partitioning. In *Proceedings of the 15th Ann. ACM SIGACT-SIGPLAN Symp. Principles of Programming Languages*, pages 319–329, San Diego, California, Jan 1988.

[22] M. Jimenez. *Multilevel Tiling for Non-Rectangular Iteration Spaces*. PhD thesis, Universitat Politecnica de Catalunia, 1999.

[23] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The Omega Library Interface Guide. Technical Report CS-TR-3445, CS Dept., Univ. of Maryland, College Park, Mar 1995.

[24] Chung-Ta King, W-H Chou, and L. Ni. Pipelined Data-Parallel Algorithms: Part II Design. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):430–439, Oct 1991.

[25] W. Li. *Compiling for NUMA Parallel Machines*. PhD thesis, Cornell Univ., Ithaca, New York, 1993.

[26] J. Ramanujam. Non-Unimodular Loop Transformations of Nested Loops. In *Supercomputing 92*, pages 214–223, Minneapolis, Nov 1992.

[27] J. Ramanujam. Beyond Unimodular Transformations. *Journal of Supercomputing*, 9(4):365–389, Oct 1995.

[28] J. Ramanujam and P. Sadayappan. Tiling Multidimensional Iteration Spaces for Multicomputers. *Journal of Parallel and Distributed Computing*, 16:108–120, 1992.

[29] W. Shang and J.A.B. Fortes. Independent Partitioning of Algorithms with Uniform Dependencies. *IEEE Trans. on Computers*, 41(2):190–206, Feb 1992.

[30] J.-P. Sheu and T.-S. Chen. Partitioning and Mapping Nested Loops for Linear Array Multicomputers. *Journal of Supercomputing*, 9:183–202, 1995.

[31] J.-P. Sheu and T.-H. Tai. Partitioning and Mapping Nested Loops on Multiprocessor Systems. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):430–439, Oct 1991.

[32] A. Sotiropoulos, G. Tsoukalas, and N. Koziris. Enhancing the Performance of Tiled Loop Execution onto Clusters using Memory Mapped Network Interfaces and Pipelined Schedules. In *Proceedings of the 2002 Workshop on Communication Architecture for Clusters (CAC'02), Int'l Parallel and Distributed Processing Symposium (IPDPS'02)*, Fort Lauderdale, Florida, Apr 2002.

[33] E. Su, A. Lain, S. Ramaswamy, D. J. Palermo, E. W. Hodges, and P. Banerjee. Advanced Compilation Techniques in the PARADIGM Compiler for Distributed Memory Multicomputers. In

Proceedings of the ACM International Conference on Supercomputing (ICS), Madrid, Spain, Jul 1995.

[34] P. Tang and J. Xue. Generating Efficient Tiled Code for Distributed Memory Machines. *Parallel Computing*, 26(11):1369–1410, 2000.

[35] P. Tsanakas, N. Koziris, and G. Papakonstantinou. Chain Grouping: A Method for Partitioning Loops onto Mesh-Connected Processor Arrays. *IEEE Trans. on Parallel and Distributed Systems*, 11(9):941–955, Sep 2000.

[36] M. Wolf and M. Lam. A Data Locality Optimizing Algorithm. In *ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI)*, Toronto, Ontario, Jun 1991.

[37] M. Wolf and M. Lam. A Loop Transformation Theory and an Algorithm to Maximize Parallelism. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):452–471, Oct 1991.

[38] J. Xue. Automatic Non-unimodular Loop Transformations for Massive Parallelism. *Parallel Computing*, 20(5):711–728, 1994.

[39] J. Xue. Communication-Minimal Tiling of Uniform Dependence Loops. *Journal of Parallel and Distributed Computing*, 42(1):42–59, 1997.

[40] J. Xue and W.Cai. Time-minimal Tiling when Rise is Larger than Zero. *Parallel Computing*, 28(6):915–939, 2002.

Biographies

Georgios Goumas received his Diploma in Electrical and Computer Engineering from the National Technical University of Athens in 1999. He is currently a PhD candidate in the School of Electrical and Computer Engineering, National Technical University of Athens. His research interests include Parallel Processing (Parallelizing Compilers, Automatic Loop Partitioning), Parallel Architectures, High Speed Networking and Operating Systems. Georgios Goumas is a recipient of the IEEE IPDPS 2001 best paper award for the paper "Minimising Completion Time for Loop Tiling with Computation and Communication Overlapping". He is a student member of the IEEE.

Maria Athanasaki received her Diploma in Electrical and Computer Engineering from the National Technical University of Athens in 2001. She is currently a PhD candidate in the School of Electrical and Computer Engineering, National Technical University of Athens. Her research interests include Parallel and Distributed Systems, Parallelizing Compilers, Dependence Analysis and High Performance Numerical Applications. She is a student member of the IEEE.

Nectarios Koziris received his Diploma in Electrical Engineering from the National Technical University of Athens (NTUA) and his Ph.D. in Computer Engineering from NTUA (1997).

He joined the Computer Science Department, School of Electrical and Computer Engineering at the National Technical University of Athens in 1998, where he currently serves as an Assistant Professor. His research interests include Computer Architecture, Parallel Processing, Parallel Architectures (OS and Compiler Support, Loop Compilation Techniques, Automatic Algorithm Mapping and Partitioning) and Communication Architectures for Clusters. He has published more than 50 research papers in international refereed journals and in the proceedings of international conferences and workshops. He has also published two Greek textbooks "Mapping Algorithms into Parallel Processing Architectures", and "Computer Architecture and Operating Systems". Nectarios Koziris is a recipient of the IEEE IPDPS 2001 best paper award for the paper "Minimising Completion Time for Loop Tiling with Computation and Communication Overlapping" (held at San Francisco, California). He is reviewer in International Journals and Conferences. He served as a Program Committee member in HiPC-2002 and CAC03 (organized with IPDPS03) Conferences, Program Committee co-Chair for both the ACM SAC 2003 and 2004 Symposiums on Applied Computing-Special Track on Parallel, Distributed Systems and Networking. He conducted research in several EU and national Research Programmes. He is a member of IEEE Computer Society, member of IEEE-CS TCPP and TCCA (Technical Committees on Parallel Processing and Computer Architecture), ACM and organized the Greek IEEE Chapter Computer Society.