The Journal of Supercomputing (2019) 75:2665–2690
https://doi.org/10.1007/s11227-018-2678-x

Insight into tiles generated by means of a correction technique

Wlodzimierz Bielecki[1] · **Piotr Skotnicki**[1]

Published online: 11 November 2018

Abstract
Well-known techniques for tiled code generation are based on the polyhedral model and affine transformations. An alternative approach to generation of tiled code is to correct original rectangular tiles defined for a loop nest by means of the transitive closure of a dependence graph instead of deriving and applying affine transformations. In this paper, we present results of an analysis of basic features of tiles generated due to correction of original rectangular tiles. We introduce procedures which allow us to recognize such features as target tile type (fixed, varied, parametric), dimensionality, size (the number of statement instances within a tile), and loop nest tileability (the percentage of statement instances that can be tiled with rectangular tiles). We consider differences between those features of tiles generated by means of affine transformations and transitive closure. We also discuss results of experiments with PolyBench benchmarks and show how differences in tiles generated with the examined approach and affine transformations affect serial tiled code performance.

Keywords Optimizing compilers · Tiling · Transitive closure · Dependence graph · Code locality

Mathematics Subject Classification 68N20 · 68N15 · 68M20 · 68R01 · 57M15 · 90C10

✉ Piotr Skotnicki
pskotnicki@wi.zut.edu.pl

Wlodzimierz Bielecki
wbielecki@wi.zut.edu.pl

[1] Faculty of Computer Science, West Pomeranian University of Technology, ul. Zolnierska 49, 71-210 Szczecin, Poland

1 Introduction

Tiling [8,11,13,15,24,30] is an iteration reordering transformation of significant importance for both improving data locality and extracting coarse-grained loop nest parallelism.

To our best knowledge, well-known tiling techniques are based on linear or affine transformations of program loops [8,11,13,15,24,30,32]. Under affine transformations, tiling is valid when there exists a band of fully permutable loops [13,30,33], i.e., when all correspondingly permuted distance vectors have nonnegative elements.

In paper [4], the authors present a novel approach for generation of tiled code for affine loop nests. It is based on correction of original rectangular tiles by means of the transitive closure of a loop nest dependence graph. That approach produces tiled code even when there does not exist any affine transformation allowing for producing a fully permutable loop nest. It breaks cycles in the inter-tile dependence graph and makes all target tiles valid under the lexicographical order of target tile enumeration.

The effectiveness of the tile correction technique used for tiling and parallelizing bioinformatics code is demonstrated in papers [18,19].

In general, corrected tiles are different from those generated by means of affine transformations. The goal of this paper is to demonstrate what are the basic features of corrected tiles such as type, size, and dimensionality. We also consider loop nest tileability—the percentage of statement instances that can be tiled with rectangular tiles in the original iteration space.

The contributions of this paper over previous work are as follows:

- presentation of basic features of corrected tiles generated by means of the transitive closure of a loop nest dependence graph such as type, size, dimensionality and how those features can be recognized by means of the introduced procedures;
- demonstration of similarities and differences of tiles generated by means of transitive closure and affine transformations;
- implementation of the presented procedures to recognize basic features of corrected tiles in the TC optimizing compiler applying the ISL library and presentation of experimental results obtained by means of that compiler for PolyBench benchmarks.

The rest of the paper is organized as follows. Section 2 contains background. Section 3 introduces types of target tiles generated by means of transitive closure and how they can be recognized. Section 4 discusses how the number of tiles within a particular group, tile size, and tile dimension can be revealed. Section 5 addresses loop nest tileability. In Sect. 6, we present features of tiles generated for loop nests from the PolyBench benchmark suite [21]. Section 7 summarizes and reviews basic features of target tiles generated by means of transitive closure and compares them with those produced by means of affine transformations. Section 8 discusses related work. Section 9 concludes the paper and outlines our plans for future work.

2 Background

In this paper, we deal with affine loop nests where, for given loop indices, lower and upper bounds as well as array subscripts and conditionals are affine functions of surrounding loop indices and possibly of structure parameters (defining loop index bounds), and the loop steps are known constants.

Given a loop nest with q statements, we transform it into its polyhedral representation, including: an iteration space IS_i for each statement S_i, $i = 1, \ldots, q$, read/write access relations (RA/WA, respectively), and global schedule S corresponding to the original execution order of statement instances in the loop nest.

The loop nest iteration space IS_i is the set of statement instances executed by a loop nest for statement S_i. An access relation maps an iteration vector I_i to one or more memory locations of array elements. Schedule S is represented with a relation which maps an iteration vector of a statement to a corresponding multidimensional timestamp, i.e., a discrete time when the statement instance has to be executed. We define a global iteration space IS as $IS = \bigcup_{i=1}^{i=q} IS_i$ and a global iteration vector I as $I = \bigcup_{i=1}^{i=q} I_i$. Further on, under I and IS, we mean the global iteration vector and global iteration space, respectively.

The positive transitive closure of a given relation R, R^+, is defined as follows:

$$R^+ = \bigcup_{i=1}^{i=\infty} R^i, \tag{1}$$

where R^i is the i-th power of R, defined inductively by $R^1 = R$, and for $i > 1$, $R^i = R \circ R^{i-1}$, where $\circ$ denotes the composition of relations.

Relation R is reflexively closed on set D if the identity relation Id_D is a subset of R. The reflexive closure of R on D is $R \cup Id_D$. The reflexive and transitive closure of R on D is:

$$R^* = R^+ \cup Id_D. \tag{2}$$

Techniques aimed at calculating the transitive closure of a dependence graph, which in general is parametric, are presented in papers [2,14,25] and they are out of the scope of this paper.

In order to compute the transitive closure of a dependence graph for loop nests examined in this paper, we have used the `isl_map_transitive_closure` function [27], the iterative approach [3], and the modified Floyd–Warshall algorithm [2]. The necessity of the usage of different algorithms is justified by our goal to get exact dependence graph transitive closure for each studied kernel and therefore to avoid any performance loss of target code unrelated to the algorithm itself. If one algorithm fails to calculate exact transitive closure, we choose another from those mentioned above.

In paper [4], a novel approach allowing for generation of valid tiles based on the transitive closure of a dependence graph is presented. Below we recap the main idea of that algorithm. First, original rectangular tiles are defined in the loop nest iteration space. Each such a tile is associated with a parametric identifier II. Then an invalid dependence target and the corresponding invalid tile are defined as follows: if there

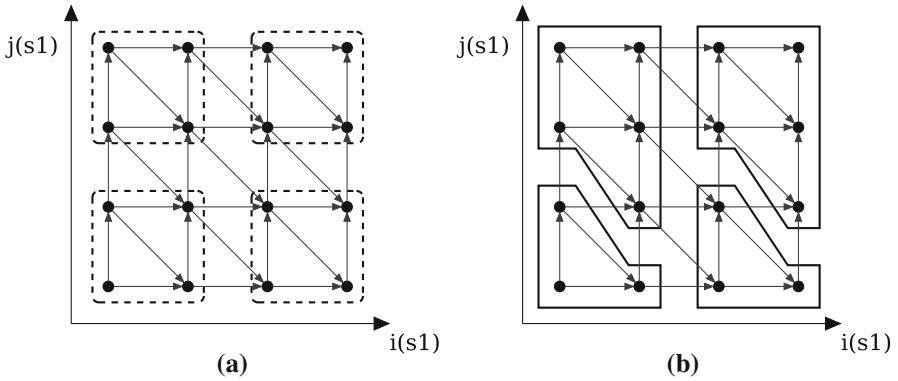

Fig. 1 Tiled iteration space for Example 1. **a** Original tiles for Example 1, **b** target tiles for Example 1

exists a direct or transitive dependence whose target belongs to a tile with identifier II while its source belongs to a tile with an identifier lexicographically greater than II, then those dependence target and tile are invalid. Each invalid tile should be corrected so that invalid dependence targets are relocated to lexicographically greater tiles.

In order to present that technique, let us consider the following example.

Example 1

```
  for (i = 1; i <= 4; ++i)
    for (j = 1; j <= 4; ++j)
S1:  A[i][j] = A[i][j+1] + A[i+1][j] + A[i+1][j-1];
```

In this paper, we present sets and relations using the iscc syntax [26]. The following relation describes all the dependences in the presented loop nest:

$$R := \{ \mathrm{S1}[i, j] \to \mathrm{S1}[1 + i, j'] \mid 0 < i \leq 3 \wedge j \leq 4 \wedge j' \geq -1 + j \wedge 0 < j' \leq j \}$$
$$\cup \{ \mathrm{S1}[i, j] \to \mathrm{S1}[i, 1 + j] \mid 0 < i \leq 4 \wedge 0 < j \leq 3 \}.$$

Figure 1a shows the dependence graph for Example 1. The squares, depicted with dashed lines, represent sets of iterations forming original rectangular tiles $T1$, $T2$, $T3$, and $T4$ of the size 2×2.

It is obvious that scanning those tiles and iterations within each tile in lexicographic order is incorrect because of the violation of the valid execution of dependent iterations (to honor a dependence, we should first execute the source of this dependence, then its target). For example, iteration (2, 2)—the target of the dependence $(1, 3) \to (2, 2)$ would be executed before iteration (1, 3)—the source of this dependence. To cope with such a problem, we correct the content of the tiles as follows. We remove iteration (2, 2) from $T1$ and add it to $T2$, remove iteration (4, 2) from $T3$ and add it to $T4$. After these changes, we get target tiles $T1\_VLD$, $T2\_VLD$, $T3\_VLD$, and $T4\_VLD$ presented in Fig. 1b. Now scanning tiles $T1\_VLD$, $T2\_VLD$, $T3\_VLD$, $T4\_VLD$ and iterations within each tile in lexicographic order is valid.

A formal algorithm of generation of target tiles based on transitive closure and the proof of the correctness of that algorithm is presented in paper [4].

In this paper, we deal with the following sets generated by means of the tiling algorithm: (i) set $TILE(II)$ including statement instances of the original rectangular tile whose identifier is II; (ii) set II_SET comprising all tile identifiers; (iii) set $TILE\_GT(II)$ including elements of all of the tiles whose identifiers are greater than identifier II; and (iv) set $TILE\_VLD(II)$ representing target corrected tiles. Hereinafter, we present an analysis of only target tiles and the corresponding serial tiled code because for a given fixed original tile size, the considered algorithm generates unique tiles and unique serial code. Parallel tiled code can be generated by means of many different approaches: assuming that a tile is a macro-atomic statement, all known parallelization algorithms can be applied to serial tiled code: techniques based on affine transformations and/or those based on transitive closure, for example, see paper [20]. In general, each technique can generate a particular parallel tiled code different from that returned by another technique, i.e., parallel tiled code is not unique. Parallel code generation is out of the scope of this paper.

The next three sections present three algorithms that allow us (i) to recognize types of target tiles; (ii) to calculate the number of tiles within a particular category of tiles and tile dimensions; (iii) to determine loop nest tileability. Those algorithms are implemented in the TC[1] optimizing compiler, which enables the user to perform one, two, or all three algorithms and collect statistics about target tiles generated with that compiler by means of the tile correction technique.

3 Types of target tiles

In this section, we introduce the following types of target tiles: fixed, varied, parametric, and fixed boundary. Subsequently, we show how those types can be recognized. Let us remind that under the examined algorithm, all original tiles are rectangular and their size is defined by input values $b_1, b_2, \ldots, b_d$, where d is the loop nest depth.

To check whether the correction of original tiles is required, we have to calculate relation R^+, sets $TILE$ and $TILE\_GT$ according to the examined algorithm and check whether the condition below is satisfied:

$$R^+(TILE\_GT) \cap TILE = \emptyset. \tag{3}$$

If condition (3) is true, this means that all statement instances belonging to set $TILE$ are valid, i.e., the set $R^+(TILE\_GT)$ does not contain any invalid dependence target within set $TILE$; therefore, target tiles are the same as original ones.

If condition (3) is not true, this means that some points (at least one) within set $TILE$ are invalid dependence targets and they will be eliminated from set $TILE$ and moved to particular lexicographically greater tiles so that all target tiles become valid. As a consequence, target tiles will differ in shape from original ones; in general, they can be non-rectangular. In such a case, the discussed algorithm can generate *fixed*, and/or *varied*, and/or *parametric* tiles. A fixed tile means that its size is constant—it does not depend on neither indices $ii_1, ii_2, \ldots, ii_d$, defining a tile identifier, nor

[1] http://tc-optimizer.sourceforge.net.

parametric upper loop index bounds. The size of varied tiles depends on indices ii_1, $ii_2, \ldots, ii_d$, defining tile identifiers, but it does not depend on parametric upper loop index bounds. The size of a parametric tile is defined by parametric upper loop index bounds.

To recognize target tile types for given upper loop index bounds, we calculate by means of the Barvinok library [26] the cardinality of set $TILE\_VLD$ (card($TILE\_VLD$)) represented with a piecewise quasipolynomial, i.e., a subdivision of one or more spaces with a quasipolynomial associated with each cell in the subdivision. Each quasipolynomial is of the following form $\{ expression : domain \}$, where $expression$ defines the number of elements of a tile group, while $domain$ defines tile identifiers of this group. Subsequently, we analyze the *expression* part of each quasipolynomial—if it involves parameters, representing upper loop index bounds, we conclude that the number of statement instances included in a tile is dependent on upper loop index bounds, i.e., the tile is *parametric*. In case of the presence of symbolic constants representing tile identifiers ($ii_1, ii_2, \ldots, ii_d$), we categorize the tile as *varied*. If both types of symbols are present—parametric upper loop index bounds and tile identifiers—we refer to corrected tile as *parametric and varied*. If an analyzed expression does not depend on any variable, the corresponding tile is *fixed*.

Within the category of fixed tiles, we mark up *fixed boundary* tiles, their size is fixed, and some their points belong to the boundaries of the loop nest iteration space. Expressions representing the size of such tiles include upper loop bounds as parameters, but the number of points within them is equal or less than that of a non-boundary original rectangular tile.

Below, we illustrate all of the aforementioned cases. Let us consider the following loop nest.

Example 2

```
   for (i = 0; i <= N; ++i) {
     for (j = 0; j <= N; ++j) {
S1:    A[i][j] = A[i][j+1];
     }
S2: B[i] = A[i][N] + A[i+1][0];
   }
```

Figure 2 presents the dependence graph, original tiles $T1$, $T2$, …, $T6$ of the size 2×2 (shown with dashed lines), and target tiles $T1\_VLD$, $T2\_VLD$, …, $T6\_VLD$ (shown with solid lines) produced by means of the examined algorithm for Example 2 when $N = 3$. It is worth noting that the structure of the dependence graph for the examined loop nest is a chain: each next iteration depends on the previous one, i.e., only the serial (lexicographical) order of iteration enumeration is valid.

To recognize tile types in a formal way, we calculate card($TILE\_VLD$) which is as follows:

$$\text{card}(TILE\_VLD) := [N, ii, ii', jj] \rightarrow \{ 2 \mid ii' = 0 \wedge ii \geq 0 \wedge 2ii \leq N \wedge jj \geq 0 \wedge 2jj \leq -2 + N \}$$

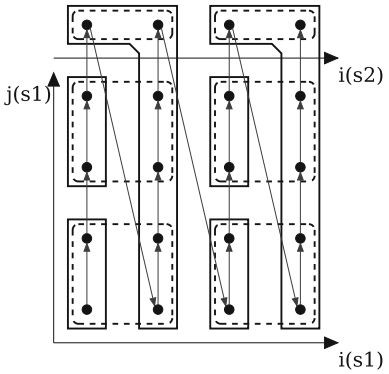

Fig. 2 Dependences and tiles for Example 2 when $N = 3$

$$\cup[N, ii, ii', jj] \rightarrow \{(3+N) \mid ii' = 1 \wedge jj = 0 \wedge ii \geq 0 \wedge 2ii < N\}$$
$$\cup[N, ii, ii', jj] \rightarrow \{(1+N-2jj) \mid ii' = 0 \wedge ii \geq 0 \wedge 2ii \leq N \wedge N-1 \leq 2jj \leq N\}$$
$$\cup[N, ii, ii', jj] \rightarrow \{1 \mid 2ii = N \wedge ii' = 1 \wedge jj = 0 \wedge N \geq 0\}.$$

Analyzing the domains of the quasipolynomials, we conclude that there can exist four groups of tiles: (i) two groups of *fixed* tiles (containing 2 and 1 statement instances, respectively); (ii) one group of *parametric* tiles whose number of statement instances is expressed as $3 + N$; (iii) one group of *parametric and varied* tiles each including $1 + N - 2jj$ statement instances.

Below we present a procedure allowing us to recognize what are the types of target tiles in a formal way.

Summing up, we may conclude that the considered algorithm relying on the dependences available in the loop nest can generate fixed and/or varied, and/or parametric tiles, all of which can be recognized by applying Algorithm 1.

4 The number of tiles within a particular group and tile dimensions

In this section, we demonstrate how to calculate the number of tiles within a particular group, tile dimensions, and the percentage of subspaces occupied with target tiles of a particular group.

We start with Example 2 presented in the previous section. Determining the number of tiles of each category and statement instances thereof is possible only for known loop nest upper bounds. Let us analyze the result of the cardinality operator applied to set $TILE\_VLD$ for a fixed value of $N = 3$:

$$\begin{aligned}\text{card}(TILE\_VLD) := &[ii, ii', jj] \rightarrow \{2 \mid ii' = 0 \wedge jj = 0 \wedge 0 \leq ii \leq 1\} \\ &\cup [ii, ii', jj] \rightarrow \{6 \mid ii' = 1 \wedge jj = 0 \wedge 0 \leq ii \leq 1\} \\ &\cup [ii, ii', jj] \rightarrow \{2 \mid ii' = 0 \wedge jj = 1 \wedge 0 \leq ii \leq 1\},\end{aligned}$$

Algorithm 1 Recognition of types of target tiles generated by means of the examined algorithm.

Input: A loop nest of depth d; fixed values $b_1, b_2, \ldots, b_d$, defining an original rectangular tile size; relation R describing all the dependences available in the loop nest; values of upper loop index bounds.
Output: Tile types generated by the examined algorithm.
Method:

1. Generate relation R^+ and sets $TILE$, $TILE\_GT$ (according to the examined algorithm) and check whether $R^+(TILE\_GT) \cap TILE = \emptyset$; if so, print: "all target tiles are fixed and the same as original ones because all points within each original tile are valid"; the end of the procedure.
2. Compute card($TILE\_VLD$). For each quasipolynomial of card($TILE\_VLD$) whose expression we denote as $expr$, perform the analysis below:
   - If the expression of a quasipolynomial is a constant, then print: "there is a group of fixed target tiles each including $expr$ statement instances."
   - If the expression of a quasipolynomial includes variables defining tile identifiers, but it does not depend on parametric upper loop index bounds, then print: "there is a group of varied tiles each including $expr$ statement instances."
   - If the expression of a quasipolynomial includes parameters defining upper loop index bounds, then print: "there is a group of parametric tiles each including $expr$ statement instances."
   - If the expression of a quasipolynomial includes both parameters defining upper loop index bounds and variables representing tile identifiers, then print "there is a group of parametric and varied tiles"; the end of the procedure.

where variables ii, jj define tile identifiers, while ii' specifies the statement whose instances are included into the tile with identifier $I = (ii, jj)^T$ ($ii' = 0$ for S1, $ii' = 1$ for S2). Taking into account the results of applying Algorithm 1 to Example 2, the above union of sets indicates that: (i) there exists a group of fixed tiles each including 2 iterations; their identifiers are defined with the constraints: $ii' = 0 \wedge jj = 0 \wedge 0 \leq ii \leq 1$; (ii) there exists a group of parametric tiles (that set corresponds to the quasipolynomial $3 + N$) each including 6 iterations; their identifiers are defined with the constraints: $ii' = 1 \wedge jj = 0 \wedge 0 \leq ii \leq 1$; (iii) there exists a group of parametric and varied tiles each comprising 2 iterations; their identifiers are defined with the constraints: $ii' = 0 \wedge jj = 1 \wedge 0 \leq ii \leq 1$; (iv) the group of fixed tiles, each including 1 iteration, is absent for $N = 3$.

In order to reveal the number of tiles within each group, we intersect set II_SET comprising tile identifiers with a set whose constraints are represented with the domain of the corresponding quasipolynomial.

By applying the constraints of the first quasipolynomial, we obtain the following modified set of tile identifiers:

$$II\_SET \cap \{\,[ii, ii', jj] \mid 0 \leq ii \leq 1 \wedge ii' = 0 \wedge jj = 0\,\} = \{\,[0, 0, 0], [1, 0, 0]\,\}.$$

We therefore conclude that there are two fixed tiles containing two statement instances. In order to discover the dimensionality of those tiles, we compute set $TILE\_VLD(II)$ for each element of the set above:

$$TILE\_VLD(\{\,[0, 0, 0]\,\}) = \{\,[0, 0, 0], [0, 0, 1]\,\},$$
$$TILE\_VLD(\{\,[1, 0, 0]\,\}) = \{\,[2, 0, 0], [2, 0, 1]\,\}.$$

Using the inverse of scheduling relation S returned by the Polyhedral Extraction Tool [29] for Example 2, S^{-1}, we discover that the tiles contain statement instances $\{\,\mathrm{S1}[0,0], \mathrm{S1}[0,1]\,\}$ and $\{\,\mathrm{S1}[2,0], \mathrm{S1}[2,1]\,\}$, respectively. Next, we compute the distance between the maximum and minimum values for each position of the tuples and add to each result 1; for both tiles, we obtain the size 1×2; hence, we conclude that the tiles are in one dimension.

The procedure is repeated for the remaining groups. For parametric tiles, we obtain the following set of tile identifiers:

$$II\_SET \cap \{\,[ii, ii', jj] \mid 0 \le ii \le 1 \wedge ii' = 1 \wedge jj = 0\,\} = \{\,[0,1,0], [1,1,0]\,\},$$

and the corresponding tiles:

$$\begin{aligned} TILE\_VLD(\{\,[0,1,0]\,\}) &= \{\,[1,0,3], [1,0,2], [1,0,1], [1,1,0], [0,1,0], [1,0,0]\,\} \\ &= \{\,\mathrm{S1}[1,3], \mathrm{S1}[1,2], \mathrm{S1}[1,1], \mathrm{S2}[1], \mathrm{S2}[0], \mathrm{S1}[1,0]\,\}, \\ TILE\_VLD(\{\,[1,1,0]\,\}) &= \{\,[3,0,3], [3,0,2], [3,0,1], [3,1,0], [2,1,0], [3,0,0]\,\} \\ &= \{\,\mathrm{S1}[3,3], \mathrm{S1}[3,2], \mathrm{S1}[3,1], \mathrm{S2}[3], \mathrm{S2}[2], \mathrm{S1}[3,0]\,\}. \end{aligned}$$

It is worth noting that each tile above comprises instances of statements S1 and S2. That is, we conclude that in the global iteration space, each tile is composed of two *subtiles*: one of the size 1×4 containing instances of statement S1 and one of the size 2×1 containing instances of statement S2.

Now, let us analyze the content of parametric and varied tiles with the number of statement instances equal to $1 + N - 2jj$. Carrying out the above procedure again, we obtain:

$$\begin{aligned} &II\_SET \cap \{\,[ii, ii', jj] \mid 0 \le ii \le 1 \wedge ii' = 0 \wedge jj = 1\,\} = \{\,[0,0,1], [1,0,1]\,\}, \\ &TILE\_VLD(\{\,[0,0,1]\,\}) = \{\,[0,0,2], [0,0,3]\,\} = \{\,\mathrm{S1}[0,2], \mathrm{S1}[0,3]\,\}, \\ &TILE\_VLD(\{\,[1,0,1]\,\}) = \{\,[2,0,2], [2,0,3]\,\} = \{\,\mathrm{S1}[2,2], \mathrm{S1}[2,3]\,\}. \end{aligned}$$

Analyzing statement instances included in the above tiles, we again conclude that these are 1-D tiles of the size 2×1. Because the width of such a tile in each of its dimensions does not exceed that of the original one of size (2×2), we reclassify the tile as *fixed boundary*.

Eventually for $N = 3$, we summarize the collected data as follows:

- the examined tiling algorithm divides the iteration spaces into 6 tiles;
- four tiles, classified as *fixed* and *fixed boundary*, make up 66.6% of all tiles, with a total of eight statement instances (40% of all statement instances are included in those tiles); each such a tile is of the size 1×2 and includes instances of statement S1 (see Fig. 2);
- two tiles, recognized to be *parametric*, constitute 33.3% of all tiles, with a total of twelve statement instances (60% of all statement instances are included in those tiles); each such a tile consists of a subtile of the size 1×4 which includes four instances of statement S1 and a subtile of the size 2×1 which includes two instances of statement S2 (see Fig. 2);

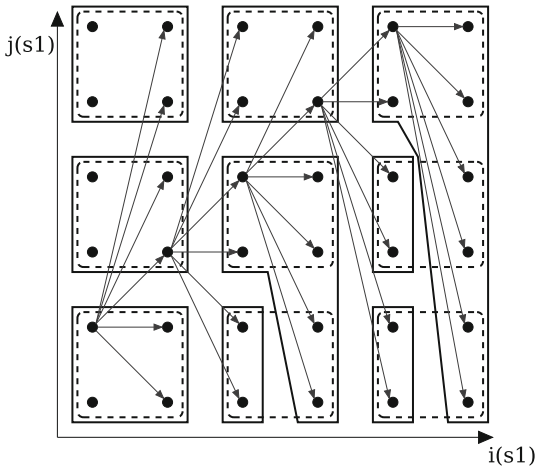

Fig. 3 Dependences and tiles for Example 3 when $N = 6$

Let us now consider the following loop nest.

Example 3

```
  for (i = 1; i <= N; ++i)
    for (j = 1; j <= N; ++j)
S1:  A[i][j] = A[i-1][i];
```

For this example, card($TILE\_VLD$) is the following.

$$
\begin{aligned}
&\text{card}(TILE\_VLD)\\
&= [N, ii, jj] \rightarrow \{\,(4 + 2 * ii) \mid jj = ii \wedge ii > 0 \wedge 2ii \leq -3 + N\,\}\\
&\cup [N, ii, jj] \rightarrow \{\,4 \mid (ii \geq 0 \wedge jj > ii \wedge 2jj \leq -3 + N) \vee (ii = 0 \wedge jj = 0\\
&\wedge N \geq 3\,\}\\
&\cup [N, ii, jj] \rightarrow \{\,(2 + N) \mid 2ii = -2 + N \wedge 2jj = -2 + N \wedge N \geq 4\,\}\\
&\cup [N, ii, jj] \rightarrow \{\,(2 * N - 4 * jj) \mid -2 + N \leq 2jj < N \wedge ((ii \geq 0 \wedge jj > ii)\\
&\vee (ii = 0 \wedge N \geq 2 \wedge jj \leq 0))\,\}\\
&\cup [N, ii, jj] \rightarrow \{\,2 \mid 2ii < N \wedge 0 \leq jj < ii\,\}\\
&\cup [N, ii, jj] \rightarrow \{\,(N - 2 * jj) \mid 2ii = -1 + N \wedge N > 0 \wedge -2 + N \leq 2jj < N\,\}.
\end{aligned}
$$

Figure 3 illustrates dependences and tiles for Example 3 when $N = 6$. Original tiles are marked with dashed lines while target tiles are depicted with solid ones.

In order to discover the number of parametric tiles each including $2 + N$ points for $N = 6$, we calculate the following set:

$$
\begin{aligned}
&II\_SET \cap \{\,[ii, jj] \mid 2ii = -2 + N \wedge 2jj = -2 + N \wedge N \geq 4\,\}\\
&\quad = \{\,[ii, jj] \mid 2ii = -2 + 6 \wedge 2jj = -2 + 6 \wedge 6 \geq 4\,\}\\
&\quad = \{\,[2, 2]\,\}.
\end{aligned}
$$

Analyzing the obtained set, we conclude that there exists a single parametric tile with identifier $II = [2, 2]$. It includes the following iterations:

$$TILE\_VLD(\{[2, 2]\}) = \{[5, 5], [5, 6], [6, 1], [6, 2], [6, 3], [6, 4], [6, 5], [6, 6]\}.$$

We can observe that the statement instances are included in the rectangular area of the size 2×6. However, that area is not fully covered by statement instances. We will therefore refer to it as a 2×6 tile, *covered in* 67% statement instances.

Algorithm 2 presents in a formal way how to calculate the sizes and dimensions of tiles as well as the percentage of subspaces occupied with particular types of tiles provided that upper loop index bounds are constants. If those bounds are parameters, then the procedure can generate very messy target results, impractical for use. In general, a complex description of a resulting set $TILE\_VLD$ also makes the application of the cardinality operator a computationally intensive procedure, taking up to several minutes to obtain a result. Once a set of quasipolynomials representing tile sizes is computed, the rest computations are of linear complexity. So the complexity of Algorithms 1, 2, 3 is defined with the complexity of the card operator calculation.

Algorithm 2 Calculating tile sizes, tile dimensions, and the percentage of the iteration space occupied by each tile group.

Input: Sets $TILE\_VLD$, II_SET, and IS returned by the discussed algorithm, vector SYM whose elements are constants representing upper loop index bounds.
Output: Tile sizes, tile dimensions, and the percentage of the iteration space occupied by each tile group.
Method:

1. Substitute all the parameters representing upper loop index bounds in sets $TILE\_VLD$, II_SET, and IS with the corresponding values contained in vector SYM.
2. Calculate the number of statement instances in the loop nest domain, N_INST, as card(IS), where IS is the set representing the global loop nest iteration space.
3. Compute the result of applying the cardinality operator to set $TILE\_VLD$, card($TILE\_VLD$).
4. For each quasipolynomial $Q \in$ card($TILE\_VLD$), carry out the following steps:

   4.1. Extract the domain of Q, D, representing the constraints of the corresponding tile group, and intersect the set whose constraint is D with set II_SET. For each element II of so obtained set, carry out the following steps:

      4.1.1. Compute set $TILE\_VLD(II)$.
      4.1.2. Compute the size of a tile, $SIZE$, as the number of statement instances included in set $TILE\_VLD(II)$ as card($TILE\_VLD$).
      4.1.3. For each element i_k, $k = 1, 2, \ldots, d$, of the tuple $[i_1, i_2, \ldots, i_d]$ of set $TILE\_VLD(II)$ compute the minimum and maximum values, $i_{k\,\min}$ and $i_{k\,\max}$, respectively.
      4.1.4. Compute value A as:
      $(i_{1\,\max} - i_{1\,\min} + 1) \times (i_{2\,\max} - i_{2\,\min} + 1) \times \ldots \times (i_{d\,\max} - i_{d\,\min} + 1)$.
      4.1.5. Compute the coverage of statement instances in the tile, C, as:
      $C = 100 \times$ card($TILE\_VLD(II)$)$/A$.
      4.1.6. Calculate the percentage of the iteration space occupied by the tile, P, as:
      $P = 100 \times$ card($TILE\_VLD(II)$)$/N\_INST$.

Algorithm 3 Computing the tileability of a loop nest.

Input: A loop nest whose upper loop index bounds are constants.
Output: Percentage of tileable loop nest statement instances.
Method:

1. Calculate the number of all the statement instances of the loop nest, N_INST, as $N\_INST = \text{card}(IS)$, where IS is the global loop nest iteration space.
2. Calculate set $INVALID$ as $INVALID = R^{+}(TILE\_GT) \cap TILE$.
3. If set $INVALID$ is empty, then the loop nest tileability is 100%, the end.
4. Calculate sets of problematic statement instances:
   $PROBLEMATIC = R^{*}(INVALID)$,
   $PROBLEMATIC\_INV = (R^{+})^{-1}(INVALID)$.
5. Calculate the number of problematic statement instances:
   $N\_PROB = \text{card}(PROBLEMATIC)$,
   $N\_PROB\_INV = \text{card}(PROBLEMATIC\_INV)$.
6. Calculate the percentage of tileable statement instances:
   $TILEABILITY\_BEFORE = 100 \times (N\_INST - N\_PROB)/N\_INST$,
   $TILEABILITY\_AFTER = 100 \times (N\_INST - N\_PROB\_INV)/N\_INST$.

5 Loop nest tileability

In this section, we address the following question: what is the percentage of the iteration space that can be tiled with original rectangular tiles, i.e., without correcting them. Hereinafter, we will refer to this property as *tileability*.

In order to compute the tileability of a loop nest, we form original rectangular tiles. Next, we remove all invalid dependence targets and all of the statement instances transitively dependent on those targets. Let us refer to such statement instances as *problematic* ones. The instances that are neither the sources of invalid dependence targets nor those dependent on any of problematic statement instances will be referred to as *non-problematic*.

For the purpose of demonstrating the above concepts, let us consider the following loop nest.

Example 4

```
  for (i = 0; i <= 5; ++i)
    for (j = 0; j <= 5; ++j)
S1:  B[i][j] = B[i+1][5-j];
```

Using the Polyhedral Extraction Tool [29], we obtain the following polyhedral model for the above loop nest:

$$IS := \{\, \mathrm{S1}[i, j] \mid 0 \le i \le 5 \wedge 0 \le j \le 5 \,\};$$
$$R := \{\, \mathrm{S1}[i, j] \rightarrow \mathrm{S1}[1 + i, 5 - j] \mid 0 \le i \le 4 \wedge 0 \le j \le 5 \,\}.$$

Figure 4 illustrates dependences and tiles for Example 4; original tiles are marked with dashed lines.

The tileability of this loop nest is the percentage of non-problematic instances within the entire iteration space. We start with computing the number of statement instances, N_INST, in the loop nest domain:

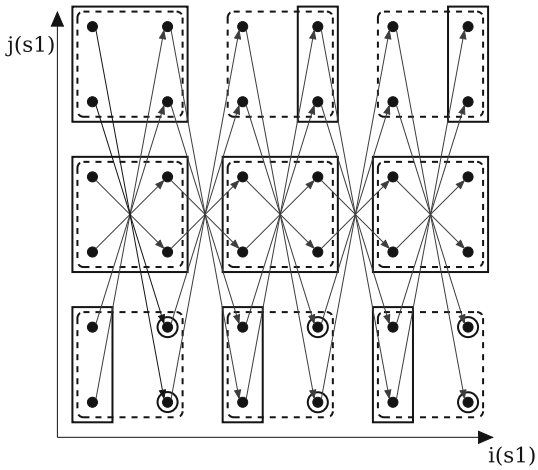

Fig. 4 Dependences and tiles for Example 4

$$N\_INST := \text{card}(IS) = 36\,.$$

Subsequently, we apply rectangular tiling to obtain set $TILE$ and the corresponding set $TILE\_GT$:

$$\begin{aligned} TILE := [ii, jj] \rightarrow \{[i, j] \mid\ & 0 \leq ii \leq 2 \wedge 0 \leq jj \leq 2 \wedge i \geq 2ii \wedge 0 \leq i \leq 5 \wedge \\ & i \leq 1 + 2ii \wedge j \geq 2jj \wedge 0 \leq j \leq 5 \wedge j \leq 1 + 2jj\,\}, \\ TILE\_GT := [ii, jj] \rightarrow \{[i, j] \mid\ & jj \geq 0 \wedge 2ii \leq i \leq 5 \wedge j \leq 5 \wedge ((ii \geq 0 \wedge \\ & jj \leq 2 \wedge i \geq 2 + 2ii \wedge j \geq 0) \vee (0 \leq i \leq 1 + 2ii \wedge \\ & j \geq 2 + 2jj))\,\}. \end{aligned}$$

Let us remind that set $TILE\_GT(II)$ consists of all the statement instances included in the tiles whose identifiers are lexicographically greater than II. Following this observation, we conclude that set, $INVALID$, including invalid dependence targets that inhibit the rectangular tiling, can be calculated as follows:

$$INVALID = R^{+}(TILE\_GT) \cap TILE. \tag{4}$$

As far as Example 4 is considered, set $INVALID$ is the following:

$$INVALID := \{[1, 0], [1, 1], [3, 0], [3, 1], [5, 0], [5, 1]\}.$$

Invalid dependence targets are surrounded by circles in Fig. 4. We must therefore remove all invalid dependence targets and all the statement instances that directly or transitively depend on invalid dependence targets. For this purpose, we compute set $PROBLEMATIC$ including all *problematic* statement instances:

$$PROBLEMATIC = R^{*}(INVALID). \tag{5}$$

By applying formula (5) to set $INVALID$ calculated for Example 4, we find the statement instances contained in set $PROBLEMATIC$ presented below:

$$PROBLEMATIC := \{[1,0],[1,1],[2,4],[2,5],[3,0],[3,1],[4,4],[4,5],[5,0],[5,1]\}.$$

Eventually, we count the number of problematic statement instances, N_PROB, by means of applying the cardinality operator to set $PROBLEMATIC$, and calculate the percentage of the iteration space, occupied by problematic statement instances, $PER\_TILEABLE$, as follows.

$$N\_PROB = \text{card}(PROBLEMATIC), \tag{6}$$

$$PER\_TILEABLE = 100 \times (N\_INST - N\_PROB)/N\_INST. \tag{7}$$

As far as Example 4 is considered, the tileability of the loop nest is 72.2%, that is, 72.2% of the iteration space can be tiled without applying the tile correction technique.

It is worth noting that after excluding problematic statement instances from set $TILE$ applying the below formula:

$$TILE\_NP = TILE - PROBLEMATIC, \tag{8}$$

we can generate code scanning tiles represented with set $TILE\_NP$ and its elements (depicted with solid lines in Fig. 4) in lexicographical order followed by executing untiled sequential code iterating over the elements of set $PROBLEMATIC$.

We call such a tileability $tilebility\_before$ because tiled code should be run before serial code enumerating problematic iterations contained within set $PROBLEMATIC$.

We can invert the execution order of problematic statement instances and non-problematic tiles provided that the set of problematic iteration points is obtained by means of applying the inverse of transitive closure to set $INVALID$, i.e., we calculate set $PROBLEMATIC\_INV$ as follows.

$$PROBLEMATIC\_INV = (R^{+})^{-1}(INVALID). \tag{9}$$

Tileability calculated on the basis of set $PROBLEMATIC\_INV$, we call $tileability\_after$, because tiled code should be run after serial code enumerating problematic iterations contained within set $PROBLEMATIC\_INV$.

Algorithm 3 presents in a formal way the calculation of loop nest tileability provided that upper loop index bounds are constants.

6 Results of experiments with PolyBench kernels

In this section, we present the results of an analysis of target tile features and corresponding code performance for PolyBench benchmarks.

Table 1 Tile types within PolyBench benchmarks

Tile categories	Kernels
Fixed (no correction)	2mm, 3mm, atax, bicg, correlation, covariance, gemm, gemver, gesummv, mvt, syr2k, syrk, trmm
Fixed (corrected) or mostly fixed	cholesky, trisolv
Fixed, parametric	lu
Fixed, parametric & varied	adi, jacobi-1d, jacobi-2d, seidel-2d
Fixed, varied, parametric & varied	durbin, fdtd-2d, gramschmidt, symm
Fixed, parametric, varied, parametric & varied	doitgen, floyd-warshall, ludcmp, nussinov

The algorithm presented in [4] and procedures introduced in this paper have been incorporated into the TC optimizing compiler,[2] which utilizes the Polyhedral Extraction Tool [29] for extracting polyhedral representations of original loop nests, the Integer Set Library [27] for performing dependence analysis, manipulating sets and relations as well as generating output code, and the Barvinok library [26] for calculating set cardinality and processing its representation.

We have experimented with the PolyBench/C 4.1 [21] benchmark suite, which includes linear algebra kernels and solvers, data mining programs, stencil computations, and dynamic programming algorithms. Generated tiles were analyzed by the introduced algorithms for data sizes defined by PolyBench as *medium*, *large*, and *extra large*. Reports generated by TC as well as tiled code generated by means of TC and Pluto for all studied PolyBench kernels can be found under the *results* directory in the project repository.

TC is able to generate set $TILE\_VLD$ and a corresponding report for 28 out of 30 programs included in PolyBench. For the two remaining kernels—*heat-3d* and *deriche*—deriving set $TILE\_VLD$ or its cardinality is too computationally intensive.

Table 1 presents types of target tiles for PolyBench kernels. TC finds a total of 13 programs that does not require any correction of original rectangular tiles. For two kernels, original tiles are corrected, but target tiles are fixed or mostly fixed. For the remaining kernels, target tiles are a mix of different types.

For five chosen kernels, we compare features of generated tiles with those produced by Pluto v.0.11.4 [8]—a state-of-the-art optimizing compiler based on the Affine Transformations Framework.

Those five numerical programs were chosen so that generated target tiles belong to different types:

- programs for which the original tiling does not require correction;
- programs for which corrected tiles are categorized as only fixed or mostly fixed;
- programs for which the correction algorithm produces tiles of various categories: fixed, parametric, varied.

[2] http://tc-optimizer.sourceforge.net.

Table 2 Execution time and speedup for TC and Pluto for the studied kernels

Kernel	Original tile size	Problem size	Original time (s)	Time (s)		Speedup	
				TC	Pluto	TC	Pluto
syr2k	64	Medium	0.0315	0.0153	0.0265	**2.06**	1.19
		Large	2.8107	1.9329	3.2618	**1.45**	0.86
		Extra large	93.1386	17.8868	30.7912	**5.21**	3.02
cholesky	32	Medium	0.0107	0.0078	0.0105	**1.37**	1.02
		Large	1.1610	0.9706	0.9655	1.20	**1.20**
		Extra large	10.2771	7.9269	7.8775	1.30	**1.30**
	64	Medium	0.0107	0.0092	0.0093	**1.17**	1.15
		Large	1.1610	1.1347	1.1401	**1.02**	1.02
		Extra large	10.2771	9.1666	9.1964	**1.12**	1.12
symm	64	Medium	0.0146	0.0100	*	**1.50**	*
		Large	1.6532	1.1889	*	**1.39**	*
		Extra large	21.1922	10.3501	*	**2.05**	*
nussinov	64	Medium	0.0258	0.0219	0.0212	1.17	**1.22**
		Large	3.0717	3.0276	2.9624	1.01	**1.04**
		Extra large	52.2635	39.8301	41.6972	**1.31**	1.25
jacobi-2d	64	Medium	0.0172	0.0155	0.0172	**1.11**	1.00
		Large	1.7527	1.9976	2.3172	0.88	0.76
		Extra large	25.4783	19.0168	21.6232	**1.34**	1.18

Bold indicates greater speedup
* No result could be obtained

Table 2 summarizes execution time and speedup (here and further on, we mean under speedup the ratio of the execution time of original code to that of serial tiled one) of tiled code produced by TC and Pluto for the studied kernels.

The evaluation of tiled code performance was carried out on an Intel Xeon E5-2699 v3 processor clocked at 2.3 GHz, 32 kB L1 data cache, 256 kB L2 cache, 45 MB L3 cache, 256 GB RAM clocked at 2133 MHz. Code of both original and transformed loop nests was compiled under the Linux kernel 3.10.0 x86_64 by GCC 4.8.3 with the O3 optimization enabled.

Below, we analyze the features of those five codes in detail.

6.1 syr2k

syr2k is a basic linear algebra subprogram updating a square $N \times N$ matrix C according to the formula: $C := \alpha AB^T + \alpha BA^T + \beta C$. The kernel consists of two separate perfectly nested loop nests—one of depth 2 multiplying the elements of C by β and one of depth 3 performing the AB^T and BA^T matrix multiplications. The *syr2k* kernel serves as an example of a loop nest, for which original tiles do not require any correction, i.e., there are no invalid dependence targets in each set $TILE$, so target tiles

Table 3 Features of tiles for kernel *syr2k* (original tile size 64 × 64 × 64)

Tile category	Upper bounds					
	N = 240 M = 200		N = 1200 M = 1000		N = 2600 M = 2000	
	Tiles	Stmts	Tiles	Stmts	Tiles	Stmts
Fixed 2-D	11.25%	0.32%	5.28%	0.09%	2.88%	0.05%
Fixed 3-D	33.75%	61.13%	79.19%	88.39%	89.41%	96.12%
Fixed boundary 2-D	8.75%	0.18%	0.60%	0.01%	0.15%	< 0.01%
Fixed boundary 3-D	46.25%	38.37%	14.93%	11.51%	7.56%	3.83%
Total	80	11,577,600	6137	1,441,440,000	55,473	13,526,760,000
Invalid	0.00%		0.00%		0.00%	
Tileability before	100.00%		100.00%		100.00%	
Tileability after	100.00%		100.00%		100.00%	

are fixed and the same as the original ones. Tiling such a loop nest is straightforward—no correction of original tiles is needed; output code enumerates fixed 2-D and 3-D tiles. Due to the fact that the discussed algorithm originally applies rectangular tiling, we can also expect that the code produced by TC will be similar to that generated by Pluto.

Table 3 summarizes the features of tiles for different upper loop index bounds and the original tile size 64 × 64 × 64, extracted by means of Algorithms 1 and 2. In this and the following tables, *Tiles* and *Stmts* define the percentage of tiles and the iteration subspace occupied with tiles and all statement instances within this group of tiles, respectively. As we can see, nearly the entire loop nest domain is occupied by 3-D fixed rectangular tiles, comprising a total of 99% of all statement instances for the largest problem size.

As shown in Table 2, the tiled code leads to a significant reduction in execution time. The only difference causing visibly worse performance of the code generated by Pluto is that Pluto interchanges the two innermost loops; this does not improve overall program performance and even leads to a slowdown for certain problem sizes.

6.2 cholesky

The *cholesky* kernel implements the Cholesky–Banachiewicz algorithm. It involves an imperfectly nested loop nest of depth 3 with 4 statements. This kernel is a representative of loop nests, for which tile correction is required, but corrected tiles are fixed or mostly fixed. This depends on an original tile size.

Tables 4 and 5 present the features of target tiles for the original tile sizes 32×32×32 and 64 × 64 × 64, respectively, collected by means of Algorithms 1, 2, and 3. For the original tile size 32 × 32 × 32, there are some 3-D parametric target tiles whose number of statement instances is greater than that of original ones. For the original

Table 4 Features of tiles for the *cholesky* kernel (original tile size 32 × 32 × 32)

Tile category	Upper bounds					
	N = 400		N = 2000		N = 4000	
	Tiles	Stmts	Tiles	Stmts	Tiles	Stmts
Fixed 2-D	10.51%	0.53%	4.01%	0.14%	2.24%	0.07%
Fixed 3-D	55.91%	69.48%	87.16%	92.91%	95.46%	97.57%
Parametric 3-D	–	–	–	–	0.04%	0.02%
Fixed boundary 2-D	4.13%	0.15%	0.27%	0.01%	0.04%	< 0.01%
Fixed boundary 3-D	29.46%	29.84%	8.56%	6.94%	2.24%	2.34%
Total	533	10,746,800	45,633	1,335,334,000	341,125	10,674,668,000
Invalid	11.31%		2.31%		1.16%	
Tileability before	0.64%		0.15%		0.07%	
Tileability after	< 0.01%		< 0.01%		< 0.01%	

tile size 64 × 64 × 64, those parametric tiles became fixed boundary, i.e., the number of statement instances within such tiles is equal or less than that of original ones.

Analyzing the data in Table 4, we discover that target tiles have the following features: (i) there are 2-D and 3-D tiles, (ii) when the values of the upper bounds of the loop indices increase, the percentage of three-dimensional tiles also increases and constitutes over 90% of all tiles, (iii) most of the statement instances are located inside three-dimensional tiles, (iv) almost all tiles are either fixed or fixed boundary.

Analyzing the structure of tiles in detail,[3] we discover that the most common tile is a 3-D fixed tile of the size 32 × 32 × 32, including 32,768 instances of statement S1. The number of those tiles constitutes 31%, 79%, and 91% of all tiles for $N = 400$, $N = 2000$, and $N = 4000$, respectively. The statement instances included in each such tile access approximately 24 kB of memory, while the capacity of L1 data cache of a computer used for experiments is 32 kB. For the largest problem size, the overall number of statement instances included in these tiles is nearly 96%. Additionally, TC produces approximately 4% of fixed 2-D tiles of the size 32 × 32 fully covered with statement instances, which—due to a regular shape of tiles—allows us to expect a satisfactory performance of tiled code similar to that produced by Pluto. That is, for the *cholesky* kernel, Pluto also generates 3-D tiles and the measured speedup is practically the same for all problem sizes as that of tiled code generated by TC.

The tileability of the loop nest is very low; it ranges from 0.64% for $N = 400$ down to 0.07% for $N = 4000$.

Analyzing data in Table 5, we can see that increasing the tile size of each dimension by a factor of 2, for the largest problem bounds, we find 35,990 fixed 3-D tiles fully covered with statement instances; they constitute 79% of all tiles. Each such a tile accesses 96 kB of memory, while the capacity of L1 data cache of a computer used

[3] Detailed reports can be found at http://tc-optimizer.sourceforge.net under the *results* directory.

Table 5 Features of tiles for kernel *cholesky* (original tile size 64 × 64 × 64)

Tile category	Upper bounds					
	N = 400		N = 2000		N = 4000	
	Tiles	Stmts	Tiles	Stmts	Tiles	Stmts
Fixed 2-D	10.48%	0.42%	6.73%	0.13%	4.01%	0.07%
Fixed 3-D	39.05%	57.53%	77.02%	88.65%	87.16%	92.98%
Fixed boundary 2-D	9.52%	0.21%	0.93%	0.01%	0.27%	< 0.01%
Fixed boundary 3-D	40.95%	41.84%	15.32%	11.21%	8.56%	6.95%
Total	105	10,746,800	6480	1,335,334,000	45,633	10,674,668,000
Invalid	22.37%		4.67%		2.35%	
Tileability before	0.53%		0.14%		0.07%	
Tileability after	< 0.01%		< 0.01%		< 0.01%	

for experiments is 32 kB. This reduces tiled code locality. As shown in Table 2, the speedup of the tiled code generated for 64 × 64 × 64 original tiles is less than that of the code generated using 32 × 32 × 32 original tiles.

6.3 symm

The *symm* kernel implements a symmetric matrix multiplication of the form $\alpha AB+\beta C$ with matrix A of the size $M \times M$, and B, C being matrices of the size $M \times N$. The results of applying Algorithms 1, 2, and 3 to this benchmark are presented in Table 6.

For the *symm* kernel, Pluto v.0.11.4 does not return any affine transformation to generate tiled code. In our opinion, this fact is due to dependences that involve scalar variable $temp2$ in the *symm* kernel. Privatization of this variable allows for respecting all the dependences involved with this variable, but to our best knowledge, Pluto v.0.11.4 does not allow for privatization of variables.

TC generates tiled code even without applying the privatization technique; this code demonstrates up to 2.05 speedup. As a result of tile correction, target tiles are a mix of fixed, varied, and parametric & varied tiles of different dimensionalities, from 1-D up to 3-D.

From Table 6, we can see that for each of the problem sizes, 50% of all tiles are one-dimensional. From a detailed report returned with TC, we find that those one-dimensional tiles are 1 × 1 × 64 ones including 64 instances of statement S3. However, the total number of statement instances included in those tiles constitutes only 0.01% of all statement instances in the loop nest domain. The majority of statement instances are included in three-dimensional tiles which constitute up to 48% of all tiles, most of which are fixed (64 × 64 × 64 tiles including 262,144 instances of statement S2). However, the procedures find also relatively large parametric and varied tiles, up to the size 64 × 2600 × 1983 for $M = 2000$ and $N = 2600$, each accessing 42 MB of

Table 6 Features of tiles for the *symm* kernel (original tile size $64 \times 64 \times 64$)

Tile category	Upper bounds					
	N = 240 M = 200		N = 1200 M = 1000		N = 2600 M = 2000	
	Tiles	Stmts	Tiles	Stmts	Tiles	Stmts
Fixed 1-D	20.00%	< 0.01%	5.88%	< 0.01%	3.03%	< 0.01%
Fixed 2-D	3.75%	0.13%	0.35%	0.01%	0.09%	< 0.01%
Fixed 3-D	11.25%	24.45%	36.57%	41.25%	42.96%	46.86%
Varied 1-D	30.00%	0.02%	44.12%	0.01%	46.97%	0.01%
Varied 2-D	7.50%	0.63%	4.88%	0.72%	2.77%	0.77%
Parametric/varied 2-D	3.75%	0.39%	0.35%	0.09%	0.09%	0.05%
Parametric/varied 3-D	5.00%	64.44%	0.31%	52.36%	0.07%	50.78%
Fixed boundary 3-D	18.75%	9.93%	7.55%	5.55%	4.01%	1.53%
Total	80	9,648,000	5168	12,01,200,000	43,296	10,405,200,000
Invalid	65.49%		53.18%		51.60%	
Tileability before	< 0.01%		< 0.01%		< 0.01%	
Tileability after	34.88%		46.90%		48.43%	

data. On average, target tiles access 52 kB, 76 kB, and 78 kB of memory for each of the studied problem sizes: medium, large, and extra large, respectively.

An interesting property of the *symm* kernel is its tileability. Nearly half of the iteration space can be tiled with rectangular tiles after the removal of problematic statement instances using the inverse of the transitive closure of a dependence relation (*tileability_after*).

6.4 nussinov

The *nussinov* kernel implements Nussinov's algorithm for predicting a possible RNA structure. The exact transitive closure for this benchmark was calculated applying the iterative algorithm presented in paper [3].

Based on Table 7, we conclude that the tile correction technique is able to produce three-dimensional tiles for the Nussinov loop nest. Although the majority of tiles are fixed, they contain approximately only 0.02% of statement instances. Most of the statement instances are included in parametric & varied 3-D tiles; this leads to a variety of target tile sizes. For the studied problem sizes—$N = 500$, $N = 2500$, $N = 5500$—tiles access on average only 14 kB, 31 kB, and 35 kB of memory, respectively. However, some of the parametric & varied tiles reach the size of 2.92 MB.

For the nussinov kernel, Pluto is able to produce only 2-D tiles, while TC generates mostly 3-D target tiles both fixed and parametric. For the largest problem size, TC code speedup is 1.31 and 1.05 against original and Pluto code, respectively.

Table 7 Features of tiles for the *nussinov* kernel (original tile size 64 × 64 × 64)

Tile category	Upper bounds					
	N = 500		N = 2500		N = 5500	
	Tiles	Stmts	Tiles	Stmts	Tiles	Stmts
Fixed 1-D	49.61%	0.02%	86.15%	0.03%	93.27%	0.02%
Fixed 2-D	2.73%	< 0.01%	0.28%	< 0.01%	0.07%	< 0.01%
Fixed 3-D	–	–	0.02%	0.02%	0.07%	< 0.01%
Varied 3-D	–	–	0.52%	7.39%	–	–
Parametric 3-D	–	–	0.01%	0.40%	–	–
Parametric/varied 3-D	13.28%	99.48%	5.26%	92.10%	3.08%	99.97%
Fixed boundary 1-D	16.02%	0.01%	6.08%	< 0.01%	3.15%	< 0.01%
Fixed boundary 2-D	14.84%	0.01%	1.40%	< 0.01%	0.35%	< 0.01%
Fixed boundary 3-D	3.52%	0.48%	0.28%	0.06%	< 0.01%	< 0.01%
Total	256	21,082,750	14,096	2,610,413,750	121,299	27,759,410,250
Invalid	82.33%		96.13%		98.22%	
Tileability before	< 0.01%		< 0.01%		< 0.01%	
Tileability after	< 0.01%		< 0.01%		< 0.01%	

6.5 jacobi-2d

Table 8 summarizes the results collected for the 5-point *jacobi-2d* kernel for the original 64 × 64 × 64 tile. According to reports generated with TC, most of the statement instances (up to 90%) are included in 3-D tiles, which constitute 44% of all tiles. These tiles contain 258,048 instances of statement S1 in subspace $63 \times 188 \times 127$, and 262,144 instances of statement S2 in subspace 64 × 190 × 127. The coverage with statement instances of that tile, however, is below 20%. Out of 2-D tiles, almost all of them are fixed 1 × 64 × 64 tiles including 4096 instances of statement S1. On average, tiles access 139 kB, 188 kB, and 200 kB of memory, for each of the examined problem sizes—medium, large, extra large—respectively.

For the *jacobi-2d* kernel, both properties—*tileability_before* and *tileability_after*—yield almost the same outcome.

Pluto generates 3-D tiles for the *jacobi-2d* kernel as well; however, the speedup over untiled code gained by TC is visibly higher. Nonetheless, both compilers produce reliable positive speedup only for sufficiently large problem sizes. Optimization performed by TC and Pluto does not compensate for increased code complexity resulting from the applied transformations—additional conditional statements within the loop nest account for the slowdown visible in Table 2.

Table 8 Features of tiles for kernel *jacobi-2d* (original tile size $64 \times 64 \times 64$)

Tile category	Upper bounds					
	N = 250 TSTEPS = 100		N = 1300 TSTEPS = 500		N = 2800 TSTEPS = 1000	
	Tiles	Stmts	Tiles	Stmts	Tiles	Stmts
Fixed 2-D	28.13%	0.60%	45.35%	0.78%	47.75%	0.77%
Fixed 3-D	18.75%	24.98%	39.97%	80.67%	44.84%	89.22%
Parametric/varied 3-D	31.25%	74.02%	10.03%	18.53%	5.16%	9.98%
Fixed boundary 2-D	21.88%	0.40%	4.65%	0.02%	2.25%	0.03%
Total	64	12,300,800	7056	1,684,804,000	61,952	15,657,608,000
Invalid	83.99%		90.14%		90.11%	
Tileability before	1.00%		0.20%		0.10%	
Tileability after	0.52%		0.10%		0.05%	

7 Summarizing basic features of target tiles

In this section, we summarize basic features of tiles generated by means of the examined algorithm and compare them with those generated by means of affine transformations. An analysis of obtained results allows us to state that the following is true: (i) the tile correction technique does not require full loop permutability and always generates target code provided that calculation of the transitive closure of a loop nest dependence graph is possible, but the size of target tiles can be fixed, and/or varied, and/or parametric; a parametric tile de facto represents a subspace where fixed tiles cannot be generated, i.e., a subspace where tiling is excluded provided that the data size per a parametric tile is greater than the capacity of cache; (ii) corrected target tile origins (the lexicographically earliest points of tiles) are the same as original ones, while the shape, size, and dimension of target tiles are specified automatically to respect all the dependences available in the original loop nest; (iii) dimensionalities of corrected target tiles can vary from one to the value equal to the depth of a loop nest; (iv) in general, the number of tiles produced by means of affine transformations is greater than that generated with applying transitive closure because affine transformations may skew the iteration space, while the examined algorithm does not transform the iteration space; (v) the tile correction technique allows for producing tiled code at one run, while well-known techniques require two runs: the first one is to find affine transformations allowing for generation of fully permutable loop bands and the second one is to apply tiling transformations to those bands to produce tiled code.

Table 9 summarizes basic features of tiled code generated by the examined algorithm and affine transformations.

Table 9 Comparison of features of tiles generated by the examined algorithm and affine transformations

Tile features	Examined algorithm	Affine transformations
Dimensionality	Subject to loop nest dependences; in general, there may be a mix of tiles of different dimensions	Defined by the number of fully permutable loops in a band
Shape	There may be different tile shapes for a given loop nest	Rectangular in a transformed iteration space except for boundary tiles
Number of tiles	Defined by the original tile size	Depends on affine transformations applied; in general, there may be more tiles than those generated by the considered algorithm
Type	There may be fixed, and/or varied, or/and parametric tiles	Fixed tiles defined by constants representing tile size in a transformed iteration space

8 Related work

There has been a considerable amount of research into tiling demonstrating how to aggregate a set of loop nest iterations into tiles with each tile as an atomic macro statement, from pioneer papers [13,24,30,32] to those presenting advanced techniques [7,11,15,28].

One of the most advanced reordering transformation frameworks is based on the polyhedral model and affine transformations. This approach includes the following three steps: (i) program analysis aimed at translating high level codes to their polyhedral representation and to provide data dependence analysis based on this representation, (ii) program transformation with the aim of improving program locality and/or parallelization, for this purpose affine transformations are derived and applied, (iii) code generation [7,9,10,16,24,28]. All above three steps are present in the examined approach. But there exists the following difference in step (ii): the approach based on transitive closure does not find and use any affine function. It applies the transitive closure of a program dependence graph to correct invalid rectangular tiles.

The tiling validity condition by Irigoin and Triolet [13] requires nonnegative elements of dependence vectors. The tiling validity condition by Xue [33] checks for lexicographic nonnegativity of inter-tile dependences. Mullapudi and Bondhugula [17] demonstrate that those conditions are conservative, i.e., they miss tiling schemes for which the tile schedule is not easy to present statically. They suggest to check whether an inter-tile dependence graph is cycle-free. If not, splitting or merging problematic original tiles can be applied manually to break cycles and then form a tile schedule dynamically, i.e., at run-time. The examined approach does not require any validity condition, it is able to generate tiled code for an arbitrary affine loop nest provided that transitive closure can be calculated. However, it does not guarantee that generated code will have a higher performance than that of the corresponding original loop nest, for example, when serial execution order of statement instances in an original loop nest is only valid. An additional analysis of tiled code is required to predict

its locality and performance. The goal of the algorithms introduced in this paper and implemented in the TC compiler is to support such an analysis.

Papers [22,23] consider the usage of transitive closure for deriving loop transformations, but they do not consider any tiling.

In paper [31], the authors introduce the definition of "mostly tileable" loop nests for which classic tiling is prevented by an asymptotically insignificant number of iterations. They suggest to peel the problematic iterations of the loop nest and apply rectangular tiling to the remaining iterations. The authors demonstrate the application of their algorithm to only one code implementing Nussinov's algorithm. The scope of the applicability of that algorithm is not presented. The examined algorithm instead of peeling problematic tiles corrects them to make tiling valid. In this paper, we present a technique that utilizes the transitive closure of a dependence relation to find a set of problematic statement instances and calculate loop nest tileability. That technique is fully automatic and can be applied to an arbitrary affine loop nest.

Index set splitting [12] partitions the loop nest iteration space. This enables finding affine transformations for different partitions and can be useful to break cycles in an inter-tile dependence graph.

Papers [1,5] demonstrate how to extract coarse- and fine-grained parallelism applying different Iteration Space Slicing algorithms; however, they do not consider any tiling transformation.

Paper [4] presents the examined algorithm and the proof of the correctness of generated target code by means of this algorithm, but it does not provide any analysis of features of tiles generated.

Paper [6] presents how to generate parallel synchronization-free tiled code based on transitive closure and the application of the discussed algorithm to different real-life benchmarks, but it does not provide any analysis of features of target tiles.

Summing up, we may conclude that this paper is the first attempt to present results allowing us to recognize basic features of tiles generated by means of the discussed algorithm.

9 Conclusion

In this paper, we presented the results of an analysis of features of tiles generated by means of correction of original rectangular tiles applying the transitive closure of dependence graphs. We introduced three algorithms which allow us to recognize such features of target tiles as type (fixed/parametric/varied), size, and dimensionality as well as loop nest tileability. We discussed differences between these features for tiles generated by means of applying transitive closure and affine transformations.

We also presented the results of experiments demonstrating what are features of target tiles generated with the examined approach for real-life codes included in the PolyBench benchmark suite and how they affect serial tiled code performance.

In the future, we intend to present other algorithms based on transitive closure that will allow us to (i) use different shapes of original tiles (e.g., parallelepiped, diamond);

(ii) form target tiles automatically without using original ones; (iii) examine different techniques to run target tiles in parallel.

References

1. Beletska A, Bielecki W, Cohen A, Palkowski M, Siedlecki K (2011) Coarse-grained loop parallelization: Iteration space slicing vs affine transformations. Parallel Comput 37:479–497
2. Bielecki W (2013) Using basis dependence distance vectors to calculate the transitive closure of dependence relations by means of the Foyd-Warshall algorithm. In: Widmayer P, Xu Y, Zhu B (eds) Combinatorial Optimization and Applications. Springer International Publishing, Cham, pp 129–140
3. Bielecki W, Klimek T, Palkowski M, Beletska A (2010) An iterative algorithm of computing the transitive closure of a union of parameterized affine integer tuple relations. In: COCOA 2010: Fourth International Conference on Combinatorial Optimization and Applications. Lecture Notes in Computer Science, vol 6508/2010, pp 104–113
4. Bielecki W, Palkowski M (2016) Tiling arbitrarily nested loops by means of the transitive closure of dependence graphs. Int J Appl Math Comput Sci 26(4):919–939
5. Bielecki W, Palkowski M, Klimek T (2012) Free scheduling for statement instances of parameterized arbitrarily nested affine loops. Parallel Comput 38(9):518–532
6. Bielecki W, Palkowski M, Skotnicki P (2018) Generation of parallel synchronization-free tiled code. Computing 100(3):277–302
7. Bondhugula U et al (2008) Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In: Hendren L (ed) Compiler constructure. Lecture notes in computer science. Springer, Berlin, pp 132–146
8. Bondhugula U et al (2008) A practical automatic polyhedral parallelizer and locality optimizer. SIGPLAN Not 43(6):101–113
9. Feautrier P (1992) Some efficient solutions to the affine scheduling problem: I. one-dimensional time. Int J Parallel Program 21(5):313–348
10. Feautrier P (1992) Some efficient solutions to the affine scheduling problem: II. Multidimensional time. Int J Parallel Program 21(6):389–420
11. Griebl M (2004) Automatic Parallelization of Loop Programs for Distributed Memory Architectures. University of Passau. Habilitation thesis
12. Griebl M, Feautrier P, Lengauer C (2000) Index set splitting. Int J Parallel Program 28(6):607–631
13. Irigoin F, Triolet R (1988) Supernode partitioning. In: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '88. ACM, New York, NY, USA, pp 319–329
14. Kelly W et al (1996) Transitive closure of infinite graphs and its applications. Int J Parallel Program 24(6):579–598
15. Lim A et al (1999) An affine partitioning algorithm to maximize parallelism and minimize communication. In: In Proceedings of the 13th ACM SIGARCH International Conference on Supercomputing. ACM Press, pp 228–237
16. Lim AW, Lam MS (1994) Communication-free parallelization via affine transformations. In: 24th ACM Symposium on Principles of Programming Languages. Springer, pp 92–106
17. Mullapudi RT, Bondhugula U (2014) Tiling for dynamic scheduling. In: Fourth International Workshop on Polyhedral Compilation Techniques, Viena
18. Palkowski M, Bielecki W (2018) Parallel tiled codes implementing the Smith-Waterman alignment algorithm for two and three sequences. J Comput Biol 25(10):1106–1119
19. Palkowski M, Bielecki W (2018) Tuning iteration space slicing based tiled multi-core code implementing Nussinov's RNA folding. BMC Bioinform 19(1):12

20. Palkowski M, Klimek T, Bielecki W (2015) TRACO: an automatic loop nest parallelizer for numerical applications. In: Federated Conference on Computer Science and Information Systems
21. Pouchet LN (2015) The polyhedral benchmark suite/c4.1. http://web.cse.ohio-state.edu/~pouchet/software/polybench. Accessed 28 Dec 2017
22. Pugh W, Rosser E (1997) Iteration space slicing and its application to communication optimization. In: International Conference on Supercomputing, pp 221–228
23. Pugh W, Rosser E (1999) Iteration space slicing for locality. In: International Workshop on Languages and Compilers for Parallel Computing. Springer, pp 164–184
24. Ramanujam J, Sadayappan P (1992) Tiling multidimensional itertion spaces for multicomputers. J Parallel Distrib Comput 16(2):108–120
25. Verdoolaege S et al (2011) Transitive closures of affine integer tuple relations and their overapproximations. In: Proceedings of the 18th International Conference on Static Analysis, SAS'11. Springer, Berlin, pp 216–232
26. Verdoolaege S (2007) barvinok: user guide. Version 0.40. http://barvinok.gforge.inria.fr/barvinok.pdf. Accessed 28 Dec 2017
27. Verdoolaege S (2010) isl: an integer set library for the polyhedralmodel. In: Mathematical software—ICMS 2010. Lecture notes in computer science, vol 6327. Springer, Berlin, pp 299–302
28. Verdoolaege S, Carlos Juega J, Cohen A, Ignacio Gomez J, Tenllado C, Catthoor F (2013) Polyhedral parallel code generation for cuda. ACM Trans Arch Code Optim 9(4):54
29. Verdoolaege S, Grosser T (2012) Polyhedral extraction tool. In: Proceedings of the 2nd International Workshop on Polyhedral Compilation Techniques. Paris, France
30. Wolf ME, Lam MS (1991) A data locality optimizing algorithm. In: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation, PLDI '91. ACM, New York, NY, USA, pp 30–44
31. Wonnacott D, Jin T, Lake A (2015) Automatic tiling of "mostly-tileable" loop nests. In: 5th International Workshop on Polyhedral Compilation Techniques, Amsterdam
32. Xue J (1997) On tiling as a loop transformation. Parallel Process Lett 7(4):409–424
33. Xue J (2000) Loop tiling for parallelism. Kluwer Academic Publishers, Norwell, MA, USA