Computing and Informatics, Vol. 36, 2017, 1261–1282, doi: 10.4149/cai_2017_6_1261

View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Computing and Informatics (E-Journal - Institute of Informatics,...

PARALLEL TILED CODE GENERATION WITH LOOP PERMUTATION WITHIN TILES

Marek Palkowski, Wlodzimierz Bielecki

Faculty of Computer Science
West Pomeranian University of Technology
Zolnierska 49, 70-210 Szczecin, Poland
e-mail: {mpalkowski, wbielecki}@wi.zut.edu.pl

Abstract. An approach of generation of tiled code with an arbitrary order of loops within tiles is presented. It is based on the transitive closure of the program dependence graph and derived via a combination of the Polyhedral and Iteration Space Slicing frameworks. The approach is explained by means of a working example. Details of an implementation of the approach in the TRACO compiler are outlined. Increasing tiled program performance due to loop permutation within tiles is illustrated on real-life programs from the NAS Parallel Benchmark suite. An analysis of speed-up and scalability of parallel tiled code with loop permutation is presented.

Keywords: Optimizing compilers, tiling, loop permutation, transitive closure, dependence graph, code locality, automatic parallelization

Mathematics Subject Classification 2010: 68N20, 65Y05, 52Bxx, 97E60, 05-XX

1 INTRODUCTION

Tiling is a very important iteration reordering transformation for both improving data locality and extracting loop nest parallelism. Tiling for improving locality groups loop nest statement instances in a loop nest iteration space into smaller blocks (tiles) allowing reuse when the block fits in local memory. Tiling for parallelism increases parallel program locality and coarsens the granularity of computation that may considerably improve parallel code performance.

Loop interchange reverses the order of two adjacent loops in a loop nest. It can be applied to ensure that the elements of a multi-dimensional array are accessed in the order in which they are present in memory, improving locality of reference. For example, if a 2-D array is stored in memory in row major order, i.e., two array elements are stored adjacent in memory (their second indices are consecutive numbers), while accesses to the array elements are in a column wise manner, program locality will be poor. However, after interchanging loops, accesses to array elements will be in row major order, this leads to enhancing code locality.

Loop interchange is valid if it does not reverse the execution order of the source and destination of any dependence in the loop nest.

Loop interchange can be generalized to loop permutation by allowing more than two loops to be moved at once and by not requiring them to be adjacent.

Sometimes it is reasonable to apply both transformations, first to produce tiled code, then to permute the order of loops within each tile. This may increase tiled code locality.

To our best knowledge, well-known tiling techniques and the interchange transformation are based on linear or affine transformations [1, 2, 3, 4, 5].

In our paper [6], we presented a novel approach to generation of tiled code for affine loop nests which is based on the transitive closure of program dependence graphs. Under that approach, first, we define original rectangular tiles, if they are not valid (their lexicographical execution order does not respect original loop nest dependences), then original tiles are corrected to be valid by means of calculations based on applying transitive closure. We demonstrated that such an approach allows producing tiled code even when there does not exist a band of fully permutable loops. But the approach presented in paper [6] does not guarantee the validity of the loop permutation transformation, i.e., it does not guarantee that changing the order of loops within target tiles will respect all dependences available in the original loop nest.

In this paper, we extend the approach presented in paper [6] so that a desired order of loops is taken as the input of an algorithm and it is guaranteed that generated target code will be valid. For both generation of valid tiled code and applying the loop permutation transformation, we use the transitive closure of a loop nest dependence graph.

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

1. proposition of an algorithm of generation of valid tiled code with an arbitrary order of loops within tiles;
2. presentation of the TRACO compiler implementing the algorithm;
3. demonstration of experimental results received by means of TRACO exhibiting program performance increase due to applying both the tiling and loop permutation techniques.

The paper is organized as follows. Section 2 presents background. Section 3 discusses loop nest tiling and permutation of loops within tiles. Section 4 includes

related work. Section 5 presents results of experiments. Section 6 concludes this paper and discusses 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 indices bounds), and the loop steps are known constants.

Dependences available in the loop nest are represented with a dependence relation of the form [*input list*] $\rightarrow$ [*output list*]: *formula*, where *input list* and *output list* are the lists of variables and/or expressions used to describe input and output tuples, and *formula* describes the constraints imposed upon input and output lists and it is a Presburger formula built of constraints represented with algebraic expressions and using logical and existential operators.

In the presented algorithm, standard operations on relations and sets are used, such as intersection ($\cap$), union ($\cup$), difference (-), domain (dom R), range (ran R), relation application ($S' = R(S) : e' \in S'$ iff exists e s.t. $e \rightarrow e' \in R$, $e \in S$). The positive transitive closure for a given relation R, R^+, is defined as follows:

$$R^+ = \{e \rightarrow e' : e \rightarrow e' \in R \vee \exists e'' \text{ s.t. } e \rightarrow e'' \in R \wedge e'' \rightarrow e' \in R^+\}. \tag{1}$$

It describes which vertices e' in a dependence graph (represented by relation R) are connected directly or transitively with vertex e.

Transitive closure, $R*$, is defined as follows [10]:

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

where I is the identity relation. It describes the same connections in the dependence graph (represented by R) that R^+ does plus connections of each vertex with itself.

Techniques aimed at calculating the transitive closure of a dependence graph, which in general is parametric, are presented in papers [9, 10, 11] and they are out of the scope of this paper. It is worth to note that existing algorithms return either exact transitive closure or its over-approximation. The former means that transitive closure represents only existing dependences in the original loop nest, while the latter implies that the representation of transitive closure includes both all existing and false (non-existing) dependences. Both representations can be used in the presented algorithm but, if we use an over-approximation of transitive closure, tiled code will be not optimal: it will allow for less code locality and/or parallelization.

The paper [12] presents the time of transitive closure calculation for NPB benchmarks [16]. It depends on the number of dependence relations extracted for a loop nest and can vary from milliseconds to several minutes (in very rare cases when the number of dependence relations is equal to hundreds or thousands).

3 TILING AND PERMUTATION OF LOOPS WITHIN TILES

In this section, we recap the tiling technique presented in paper [6], then we demonstrate how that algorithm can be modified to implement the loop permutation technique to respect all original loop nest dependences, and finally present a formal algorithm.

3.1 Tiling Based on Transitive Closure

Our goal is to transform a loop nest of depth d below

```
for(i1=lb1; i1<=ub1; i1++)
  for(i2=lb2; i2<=ub2; i2++)
    ..........................
      for(id=lbd; id<=ubd; id++)
        {S}
```

to a valid tiled loop nest of the following structure.

```
for(ii1=0; b1*ii1+lb1<=ub1; ii1++)
  for(ii2=0; b2*ii2+lb2<=ub2; ii2++)
    ............................
      for(iid=0; bd*iid+lbd<=ubd; iid++)
        for(i'1=.....)
          for(i'2=.....)
             ..............................
               for(i'd=.....)
                  {S'}
```

where $i_1, i_2, \ldots, i_d$ are the original loop indices; $ii_1, ii_2, \ldots, ii_d$ are the loop indices defining the identifier of a tile; $i'_1, i'_2, \ldots, i'_d$ are the indices of the tiled loop nest; the constants $b_1, b_2, \ldots, b_d$ define the tile size; $lb_1, lb_2, \ldots, lb_d$ and $ub_1, ub_2, \ldots, ub_d$ state for the lower and upper bounds of the loop indices, respectively; $\{S\}$ and $\{S'\}$ denote the original and target loop nest statements, respectively.

A valid tiled loop nest means that all the dependences available in the original loop are respected in the tiled loop nest.

Let us consider the following working loop nest.

```
for ( i=0; i<= 3; i++)
  for (j=0; j<= 3; j++)
    c[i][j]=c[i-1][j+1];
```

Listing 1. Working loop nest

In this paper, we use the syntaxes of the Barvinok and Omega tools [13, 27] to present sets and relations. The relation below describes dependences available in the working loop nest.

```
R:= {[i,j] -> [i+1,j-1] : 0 <= i <= 2 && 1 <= j <= 3}.
```

Our paper [6] presents an algorithm allowing for tiled code generation on the basis of the transitive closure of a loop nest dependence graph. The first step in this algorithm is to define original rectangular tiles as follows.

```
TILE:=[ii,jj]->{[i,j]: 2ii <=i<= 2ii+1, 3  and 2jj <= j <= 2jj+1,
 3 and 0 <=ii and 0 <=jj}
```

where the notation $[ii, jj] - > \{\ldots\}$ means that variables ii, jj, defining tile identifiers, are parameters in the constraints of the set $\{\ldots\}$; i, j are the loop indices, "2" represents the tile side (original tiles are of the size 2×2), and "3" states for the upper bound for variables i and j. Figure 1 a) demonstrates for the working example dependences and original rectangular tiles (shown in blue) of the size 2×2.

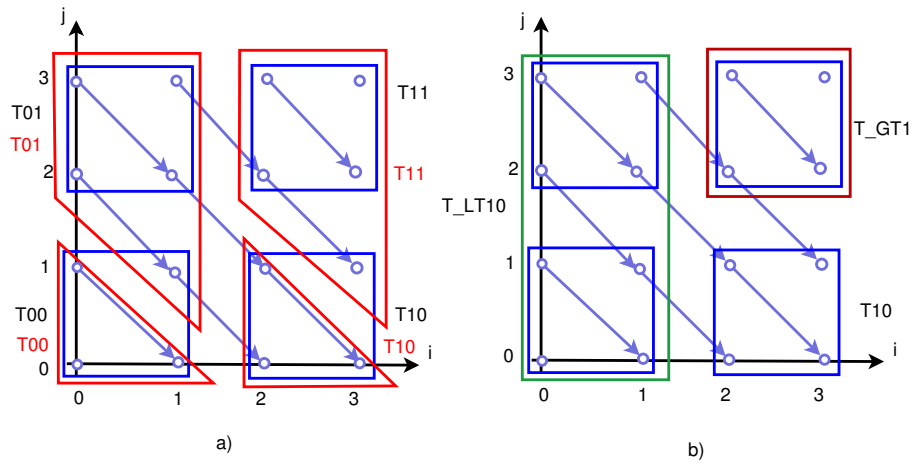

Figure 1. Dependences, original, and target tiles for the working example

The following step is to define sets $TILE\_LT$ and $TILE\_GT$ including the iterations belonging to the tiles whose identifiers are less and greater than that of $TILE$, respectively. For the working example, these sets, calculated according to the formulae presented in paper [6], are as follows.

```
TILE_LT:= {[i,j]: ii=1 and 0 <= jj <= 1 and 0 <= i <= 1 and
0 <=j <= 3} union {[i,j]: jj = 1 and 0 <= ii <= 1 and 2ii <= i
<= 2ii+1 and 0 <= j <= 1};

TILE_GT:={[i,j]: ii = 0 and 0 <= jj <=1 and 2 <= i <= 3 and
0 <= j <= 3} union {[i,j]: jj = 0 and 0 <= ii <= 1 and 2ii <= i
<= 2ii+1 and 2 <= j <= 3}.
```

Figure 1 b) presents sets *TILE_LT* and *TILE_GT* for original tile *T10*. Next, we use sets *TILE_LT*, *TILE_GT*, and the transitive closure of relation R, R^+, in the corresponding steps of the algorithm presented in paper [6], to obtain target tiles, represented by set *TILE_VLD*, as follows.

$TILE\_ITR = TILE - R^+(TILE\_GT)$,
$TVLD\_LT = (R^+(TILE\_ITR) \cap TILE\_LT) - R^+(TILE\_GT)$,

$TILE\_VLD = TILE\_ITR \cup TVLD\_LT =$

```
[ii, jj] -> { [i0, j] : jj >= 0 and j >= jj and ii >= 0 and
i0 >= 2ii and j <= 1 + 2ii + 3jj - i0 and ii <= 1 and j >=
2ii + 2jj - i0 and i0 <= 1 + 2ii and j <= 3 and j <= 1 + 2jj
and i0 <= 3 + 2ii - 2jj }.
```

Figure 1a) shows target (corrected) tiles (in red) for the working example.

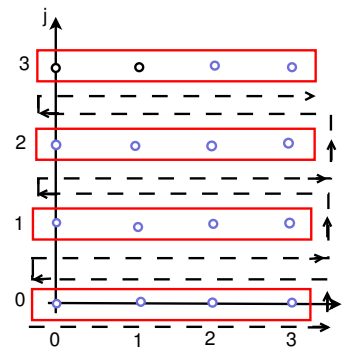

Figure 2. Changing the iteration enumeration order by means of tiling with the tile size 4×1

The algorithm presented in the paper [6] can be easily extended to arbitrarily nested loops as follows. For each loop nest statement, we have to form all sets, provided by the examined algorithm, separately and then generate tiled code using the union of extended sets representing target tiles. Details are presented in our paper [7]. Such an extension has been already implemented in TRACO (publicly available at the website `traco.sourceforge.net`).

3.2 Parallel Tiled Code Generation

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 starting from techniques based on affine transformations and ending with those based on transitive closure. The TRACO compiler,

which implements the algorithm presented in this paper, to generate parallel code applies techniques presented in the papers [14, 15] to a dependence graph whose vertices are tiles while edges point out dependences among tiles. Details are presented in the papers [7, 35, 36].

3.3 Loop Interchange

Now we are interested in applying the algorithm presented in [6] to allow for loop permutation within tiles. Our goal is to allow the user to define an arbitrary order of loops represented with indices $j_1, j_2, \ldots, j_d$, where $j_r \in \{[i_1, i_2, \ldots, i_d]\}$, $r = 1, 2, \ldots, d$, d is the loop nest depth, responsible for iteration enumeration within each tile and to guarantee that under such an order target tiled code will be valid.

Figure 2 illustrates that applying tiling with the tile size 4×1 to the 4×4 iteration space results in interchanging the order of iteration enumeration which corresponds to the lexicographical order of scanning elements of vector $\mathbf{J} = (j, i)^T$.

Analyzing Figure 1a), we can state that we cannot allow for loop interchange within each tile because this will result in invalid code since original loop nest dependences will not be respected. To cope with this problem, we can form sub-tiles of the size 2×1 within each target tile shown in red.

Figure 3a) shows such sub-tiles in green. It is worth to note that the identifiers of those sub-tiles are composed of four numbers, the first two represent target tile identifiers in the original loop nest iteration space (shown in red), while the last two numbers represent the identifiers of sub-tiles within each target tile (shown in green).

Next, we apply the algorithm, introduced in paper [6], to the subspaces representing target tiles and get target sub-tiles within each tile, Figure 3b) presents such sub-tiles in green. Now scanning target sub-tiles and iterations within each of them in the lexicographical order is valid. It is worth to note that it is not the classic loop interchange transformation within each target tile, but it is as close as possible to that.

To generate target sub-tiles within each target tile in a formal way, we first need to form set, *TILE_SUB*, which represents a parametric sub-tile within a parametric target tile. For this purpose, we introduce additional parameters ii', jj', responsible for the representation of sub-tile identifiers, and to the constraints of set *TILE_VLD*, defining target tiles, add the constraints describing a parametric tile of the size 2×1. This results in the following set where the sign "#" begins a comment.

```
TILE_SUB:=[ii,jj,ii',jj']->{[i,j]:
# the constraints below define a parametric tile of the size 2x1
# within each target tile
2ii'<=i<=2ii'+1,3 and jj'=j and j<=3 and 0<=ii' and jj'>=0 and

# the constraints below define a parametric set TILE_VLD
 (exists i0: i=i0 and jj >= 0 and j >= jj and ii >= 0 and i0 >= 2ii
```

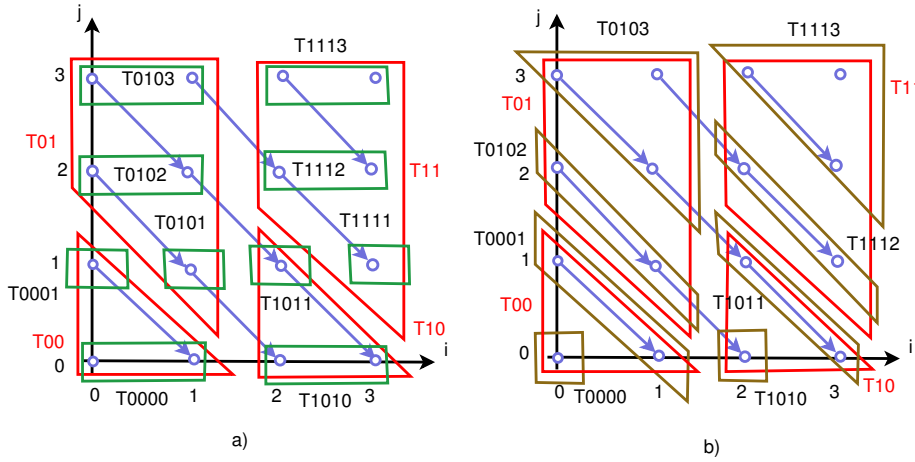

Figure 3. Dependences; original, target tiles and sub-tiles for the working example

```
 and j <= 1 + 2ii + 3jj - i0 and ii <= 1 and j >= 2ii + 2jj - i0 and
 i0 <= 1 + 2ii and j <= 3 and j <= 1 + 2jj and i0 <= 3 + 2ii - 2jj)
};
```

Next, we form sets *TILE_SUB_LT* including iterations being comprised in all the sub-tiles that are lexicographically less than that of *TILE_SUB* in the same manner as it is explained in the algorithm presented in the paper [6] except from instead of the vector of length 2, defining target tile identifiers, we use the vector of length 4 defining sub-tile identifiers. This results in the following set.

```
TILE_SUB_LT:=[ii, jj, ii', jj'] -> { [i, j]: (ii = 1 and ii' = 1
and jj <= 1 and jj' >= jj and jj' <= 1 + 2jj and jj' <= 1 + 3jj
and i <= 1 and i >= 0 and 2j >= -1 + i and j <= 3) or (ii' = ii
and ii <= 1  and ii >= 0 and jj <= 1 and jj' <= 1 + 2jj and
i <= 1 + 2ii and  j >= 2ii + 2jj - i and j >= 0 and j <= -1 + jj')
or (jj = 1 and  ii' = ii and ii >= 0 and jj' <= 3 and jj' >= 1 and
i >= 2ii and i <= 3 and j >= 0 and j <= 1 + 2ii - i) }.
```

In an analogous way, we form set *TILE_SUB_GT* and then apply the discussed algorithm to sets *TILE_SUB_LT* and *TILE_SUB_GT* to generate a set representing target sub-tiles and finally generate the target code below (see Listing 2).

This code enumerates target tiles and iterations within each target tile. The order of the iteration enumeration within each target tile is equivalent to the enumeration of sub-tiles and iterations within each sub-tile in the lexicographical order.

In general, when all elements of all dependence direction vectors of an original loop nest are non-negative, we will receive the same code as that generated with the classic loop permutation transformation, otherwise we will get valid code which is as close as possible to the code generated by the loop permutation transformation.

```
for(ii = 0; ii <= 1; ii++)
  for(jj = 0; jj <= 1; jj++)
    for(i = 2 * jj; i <= min(3, 3 * jj + 1); i++)
      for(j = 2*ii; j <= min(2*ii + 1, 2*ii - jj + i); j++){
        c[j][2 * ii + i - j]=c[j-1][2 * ii + i - j+1];
        if (jj == 1 && i == 3 && j == 2 * ii + 1)
          c[2 * ii + 1][3]=c[2 * ii][4];
      }
```

Listing 2. Tiled loop nest

3.4 Formal Algorithm

Before presenting a formal algorithm allowing for loop permutation within tiles, let us consider the loop nest of depth 3. Figure 4 presents all possible orders of loops (represented with indices i, j, k) in a 3-D tile of the size $2 \times 2 \times 2$. To satisfy a given order of loops, say i, k, j, in each 3-D tile, we have to apply first tiling within each 3-D tile with the tile size $1 \times 2 \times 2$, this will result in 2-D sub-tiles enumerated along axis i. Then within each 2-D tile, we apply tiling with the size $1 \times 2 \times 1$, this will result in 1-D tiles enumerated along axis k. To satisfy the order k, i, j, we first have to apply tiling with the tile size $2 \times 2 \times 1$; this will result in 2-D tiles enumerated along axis k. Then we apply tiling with the tile size $1 \times 2 \times 1$ which will return 1-D tiles scanned along axis i.

Figure 4 shows tile sizes which can be applied to get a given order of loops for all possible cases. The numbers in the square boxes show the order of loop nest iteration enumeration. The number of consecutive tile transformations is equal to $d - 1$, where d is the loop nest depth.

In the general case when the original order of loops is i_1, i_2, ..., i_d, the target order within each d-D tile is $j_1, j_2, \ldots, j_d$, where $j_r \in \{[i_1, i_2, \ldots, i_d]\}$, $r = 1, 2, \ldots, d$, d is the loop nest depth, and the original tile size is $b_1 \times b_2 \times \ldots \times b_d$, we apply the following scheme. We begin with the comparison of the pair j_1 and i_1, if they are different, we compare the pair j_1, i_2 and so on. As soon as the indices, say j_1, i_s, $s > 1$, are the same, we have to form (d-1)-D tiles of the size $b_1 \times b_2 \times \ldots \times b_{s-1} \times 1 \times b_{s+1} \times \ldots \times b_d$ enumerated along axis i_s. Next, we compare the pairs j_2, i_1; j_2, i_2 and so on. As soon as the indices, say j_2, i_p, are the same, supposing that $p < s$, we form within each (d-1)-D tile (d-2)-D tiles of the size $b_1 \times b_2 \times \ldots \times b_{p-1} \times 1 \times b_{p+1} \times \ldots \times b_{s-1} \times 1 \times b_{s+1} \times \ldots \times b_d$ enumerated along axis i_p. We continue this process until all pairs each including an index of the target order and one of the original order will be compared.

Algorithm 1 below presents a formal way to generate tiled code with permutation of loops within d-D target tiles. It is a modification of the algorithm presented in paper [6] and includes the following steps. The first one is the preparation of data needed to generate target code. The second step is preprocessing, it carries

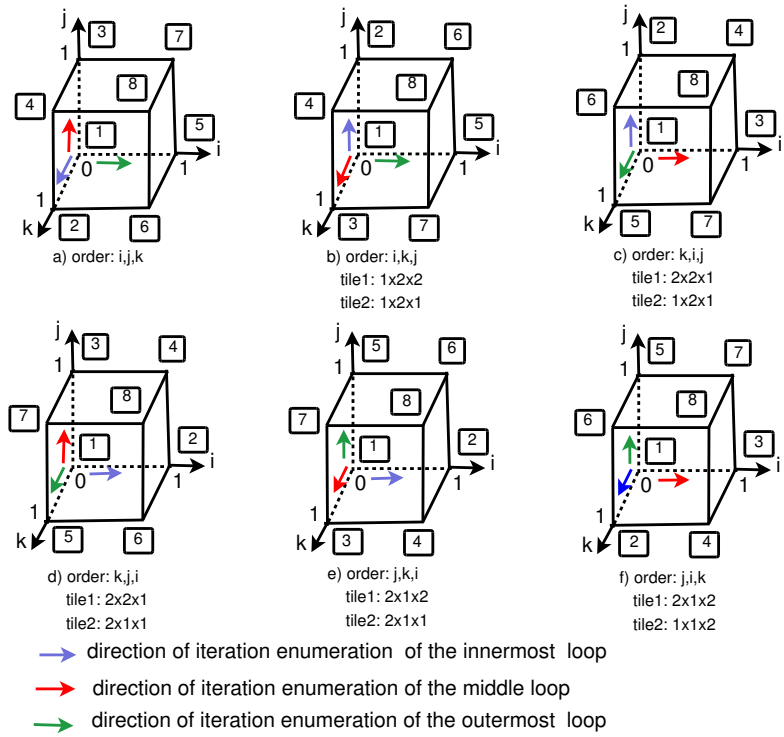

Figure 4. All possible loop orders for the loop nest of depth 3

out a dependence analysis, calculate the positive transitive closure of a dependence graph for an original loop nest, form sets to be used in the following steps, and initialize variables and sets. The last action in this step is checking whether all elements of all dependence distance vectors are non-negative, if so, this means that we can directly generate target tiled code and it is valid to permute loops within each tile according to a given input order, the proof can be found in book [8].

The third step is calculation of basic sets to be used for producing target tiles. It is based on the algorithm presented in paper [6], this algorithm is consecutively applied first to the original loop nest iteration space, then to each subspace occupied by a corresponding target tile/sub-tile. Step 4 forms sets representing valid target tiles, first for the original loop nest iteration space, then for each subspace occupied by a corresponding target tile/sub-tile. Step 5 produces a matrix responsible for tile/sub-tile size and a vector responsible for sub-tile identifiers to be used in step 3. The last step generates target code.

To justify the correctness of Algorithm 1, we take into account the following. To each loop nest iteration space (first to the original one, then to each subspace occupied by a corresponding tile/sub-tile) the algorithm presented in paper [6] is

applied. That algorithm guaranties that the lexicographical order of both tile/sub-tile enumeration and iteration enumeration in each tile/sub-tile is valid, so the target tiled code is valid.

4 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 [3, 29, 30] to those presenting advanced techniques [20, 2, 4, 21].

Advanced tiling is based on the polyhedral model. Let us remind that 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, iii) code generation [20, 22, 23, 28, 29].

All above three steps are available in the presented approach. But there exists the following difference in step ii): in the polyhedral model a (sequence of) program transformation(s) is represented by a set of affine functions, one for each statement, while 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 specific subspaces of the original loop nest iteration space. At this point of view, the program transformation step is rather within the Iteration Space Slicing Framework introduced by Pugh and Rosser [25]. In the papers [6, 7], we demonstrate that applying transitive closure instead of affine transformations does not require full loop permutability. This increases the scope of loop nests which can be tiled.

The papers [3, 30, 31, 5] are seminal works presenting the theory of tiling techniques based on affine transformations. These publications present techniques consisting of the two steps: the first one transforms an original loop nest into a fully permutable loop nest, the second one converts the fully permutable loop nest into tiled code. A loop nest is fully permutable if its loops can be permuted arbitrarily without altering the semantics of the original program. If a loop nest is fully permutable, it is sufficient to apply a tiling transformation to this loop nest [31]. The tiling validity condition by Irigoin and Triolet [3] requires non-negative elements of dependence distance vectors. The algorithm, presented in this paper, does not require full loop permutability and non-negative elements of dependence distance vectors to generate tiled code.

The papers [1, 2, 32, 4] generalize pioneer techniques and present an advanced theory on tiling implying that given a loop nest, first "time-partition constraints" are to be formed, then a solution to them has to be found. The "time-partition constraints" represent the condition that if one iteration depends on the other, then the first must be assigned to a time that is not earlier than that of the second; if they are assigned to the same time, then the first has to be executed after the second.

Algorithm 1 Tiled code generation with loop permutation within target tiles

Input: A perfect loop nest of depth d with the original order of iteration variables i_1, i_2, ..., i_d; constants $b_1, b_2, \ldots, b_d$ defining the size of an original rectangular tile; the target order of iteration variables within each tile j_1, j_2, ..., j_d, where $j_r \in \{[i_1, i_2, \ldots, i_d]\}$, $r = 1, 2, \ldots, d$.

Output: Tiled code

Method:

1 Data preparation. Form:

Vector $\boldsymbol{I}$ whose elements are original loop indices $i_1, i_2, \ldots, i_d$;

Vector $\boldsymbol{II}$ whose elements $ii_1, ii_2, \ldots, ii_d$ define the identifier of a tile;

Vectors $\boldsymbol{LB}$ and $\boldsymbol{UB}$ whose elements are lower $lb_1, lb_2, \ldots, lb_d$ and upper $ub_1, ub_2, \ldots, ub_d$ bounds of indices $i_1, i_2, \ldots, i_d$ of the original loop nest, respectively;

Vector $\mathbf{1}$ whose all d elements are equal to the value 1;

Vector $\mathbf{0}$ whose all d elements are equal to the value 0;

Diagonal matrix $\boldsymbol{B}$ whose diagonal elements are constants $b_1, b_2, \ldots, b_d$ defining the size of an original rectangular tile.

2 Preprocessing

2.1 Carry out a dependence analysis to produce a set of relations describing all the dependences in the original loop nest.

2.2 Calculate the transitive closure of the union of all the relations returned by step 2.1, R^+, applying any known algorithm, for example, [10, 9, 11].

2.3 Form set *II_SET* including the identifiers of all tiles:

$$II\_SET = \{[\boldsymbol{II}] \mid \boldsymbol{II} \geq 0 \text{ and } \boldsymbol{B}*\boldsymbol{II}+\boldsymbol{LB} \leq \boldsymbol{UB}\ \}.$$

2.4 Form set $TILE(\boldsymbol{II}, \boldsymbol{B})$ including the iterations belonging to the parametric tile defined with parameters ii_1, ii_2, ..., ii_d as follows

$$TILE(\boldsymbol{II}, \boldsymbol{B}) = [\boldsymbol{II}] \rightarrow \{[\boldsymbol{I}] j \boldsymbol{B} * \boldsymbol{II} + \boldsymbol{LB} \leq \boldsymbol{I} \leq \min(\boldsymbol{B} * (\boldsymbol{II} + \mathbf{1}) + \boldsymbol{LB} - \mathbf{1}, \boldsymbol{UB}) \text{ AND } \boldsymbol{II} \geq 0\}.$$

2.5 $k = 1$, $r = 1$, $\boldsymbol{B}_k = \boldsymbol{B}$, $\boldsymbol{II}_k = \boldsymbol{II}$, $TILE\_SUB_k(\boldsymbol{II}_k, \boldsymbol{B}_k) = TILE(\boldsymbol{II}, \mathbf{B})$, $TILE\_VLD_{k-1} = TILE(\boldsymbol{II}, \mathbf{B})$.

2.6 Check whether all elements of all dependence distance vectors are non-negative; if so, then *validity* = *TRUE*, otherwise *validity* = *FALSE*.

3 Calculation of basic sets

3.1 Form set $TILE_k(\boldsymbol{II}_k, \boldsymbol{B}_k)$ including the iterations belonging to the parametric tile defined with vector II_k as follows

$$TILE_k(\boldsymbol{II}_k, \boldsymbol{B}_k) = [\boldsymbol{II}_k] \rightarrow \{[\boldsymbol{I}] | \boldsymbol{I} \in TILE\_VLD_{k-1} \text{ AND } \boldsymbol{I} \in TILE\_SUB_k(\boldsymbol{II}_k, \boldsymbol{B}_k)\},$$ where $$TILE\_SUB_k(\boldsymbol{II}_k, \boldsymbol{B}_k) = [\boldsymbol{II}_k] \rightarrow \{[\boldsymbol{I}] j \boldsymbol{B}_k * \boldsymbol{II}_k + \boldsymbol{LB} \leq \boldsymbol{I} \leq \min(\boldsymbol{B}_k * (\boldsymbol{II}_k + \mathbf{1}) + \boldsymbol{LB} - \mathbf{1}, \boldsymbol{UB}) \text{ AND } \boldsymbol{II}_k \geq 0\}.$$

3.2 Form sets $TILE\_LT_k$ and $TILE\_GT_k$ as the union of all the tiles whose identifiers are lexicographically less and greater, respectively, than that of $TILE_k(\boldsymbol{II}_k, \boldsymbol{B}_k)$ as follows

$$TILE\_LT_k = [\boldsymbol{II}_k] \rightarrow \{[\boldsymbol{I'}] | \exists \boldsymbol{I}, \boldsymbol{II'}_k \text{ s.t. } \boldsymbol{II'}_k \prec \boldsymbol{II}_k \text{ AND } \boldsymbol{I} \in TILE_k(\boldsymbol{II}_k, \boldsymbol{B}_k) \text{ AND } \boldsymbol{I'} \in TILE_k(\boldsymbol{II'}_k, \boldsymbol{B}_k)\},$$ $$TILE\_GT_k = [\boldsymbol{II}_k] \rightarrow \{[\boldsymbol{I'}] | \exists \boldsymbol{I}, \boldsymbol{II'}_k \text{ s.t. } \boldsymbol{II'}_k \succ \boldsymbol{II}_k \text{ AND } \boldsymbol{I} \in TILE_k(\boldsymbol{II}_k, \boldsymbol{B}_k) \text{ AND } \boldsymbol{I'} \in TILE_k(\boldsymbol{II'}_k, \boldsymbol{B}_k)\},$$

4 Form set $TILE\_VLD_k$ as follows

$$TILE\_ITR_k = TILE_k(\boldsymbol{II}_k, \boldsymbol{B}_k) - R^+(TILE\_GT_k),$$
$$TVLD\_LT_k = (R^+(TILE\_ITR_k) \cap TILE\_LT_k) - R^+(TILE\_GT_k),$$
$$TILE\_VLD_k = TILE\_ITR_k \cup TVLD\_LT_k.$$

5 If *validity* = *TRUE*, then go to step 6, otherwise do

```
if (j_r ≠ i_1) then
    if (j_r ≠ i_2) then
        if (j_r ≠ i_3) then
        .....................................
            if (j_r ≠ i_d) then print "error, vector J is invalid"
            else b_d = 1, goto L
        .....................................
        else b_3 = 1, go to L
    else b_2 = 1, go to L
else b_1 = 1
```

L: if $r < d-1$ then form vector $\boldsymbol{II}_r$ whose elements ii_1^r, ii_2^r, ..., ii_d^r define identifiers of sub-tiles within the tile with identifier $\boldsymbol{II}_k$; $k = k+1$, form new vector $\boldsymbol{II}_k$ by adding the elements of vector $\boldsymbol{II}_r$ at the end of vector $\boldsymbol{II}_{k-1}$; $r = r+1$ goto step 3;

6 Code generation

6.1 Form set *TILE_VLD_EXT* by means of inserting into the first positions of the tuple of set $TILE\_VLD_k$ elements of vector $\boldsymbol{II}_k$.

6.2 Generate tiled code by means of applying any code generator scanning elements of set *TILE_VLD_EXT* in the lexicographic order, for example, CLooG [24] or the codegen function of the Omega project [19].

If there exist more than one linearly independent solutions to the time-partition constraints of a loop nest, then it is possible to apply a tiling transformation to this loop nest [4]. The presented algorithm does not require forming "time-partition constraints", it applies the transitive closure of dependence graphs to generate tiled code.

The tiling validity condition by Xue [5] checks for lexicographic non-negativity of inter-tile dependences. Mullapudi and Bondhugula [33] 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 presented algorithm allows for automatic breaking cycles in the inter-tile dependence graph and generation of tiled code statically that in general leads to higher performance of tiled code.

In the paper [34], 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 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 algorithm, presented in this paper, instead of peeling problematic tiles corrects them automatically to make them valid and can be applied to any loop nests.

The papers [14, 15] demonstrate how we can extract coarse- and fine-grained parallelism applying different Iteration Space Slicing algorithms, however they do not consider any tiling transformation.

The papers [6, 7] present algorithms based on transitive closure and the proof of the correctness of generated target code, but they do not allow for automatic loop permutation within tiles.

The paper [36] exhibits 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 consider loop permutation within tiles.

The paper [35] discuses how to form free-scheduling for tiles in code generated by means of transitive closure, but it does not allow for loop permutation within tiles.

Loop interchange is well-known reordering transformation. When all elements of all dependence distance vectors are non-negative, it is valid to permute loops within each tile according to a given input order [8]. Otherwise loop permutation may result in invalid code. The algorithm, presented in this paper, is able to produce always valid tiled code with loop permutation within tiles. When all elements of all dependence direction vectors of an original loop nest are non-negative, it will generate the same code as that generated with the classic loop permutation transformation, otherwise it will produce valid code which is as close as possible to the code generated by the loop permutation transformation.

Summing up, we may conclude that the algorithm, presented in this paper, allows for both automatic tiled code generation and loop permutation within tiles. This algorithm is to be implemented in optimizing compilers generating automatically parallel tiled code.

5 RESULTS OF EXPERIMENTS

The presented algorithm has been implemented in the optimizing compiler TRACO, publicly available at the website `traco.sourceforge.net`. For calculating the transitive closure of a loop nest dependence graph, TRACO uses the corresponding function of the ISL library [18].

To carry out experiments, we chosen the loop nests shown in the first column of Table 1 which are a subset of the NAS Parallel Benchmark suite 3.3 (NPB) [16]. We chosen codes with different structures, both perfectly (all statements are within the innermost loop) and imperfectly nested loops. The second column in Table 1 presents the original order of loops in each examined loop nest. The third column includes the order of the indices of arrays available in statements of the corresponding loop nest body; "digit" means that a corresponding index is represented by a digit. The last column demonstrates what is the permuted loop order in each tile of tiled code.

Loop Nest	**Original Order of Loops**	**Order of Array Indices**	**Permuted Order of Loops**
BT_rhs_1	k, j, i	i, j, k; digit, i, j, k	j, i, k
FT_auxfnct_2	i, k, j	j, k, i	k, j, i
LU_pintgr_2	j, i	i, j; digit, i, j, digit	i, j
SP_nivr_1	k, j, i	digit, i, j, k	j, i, k
UA_adapt_10	iz, ix, ip	ix, digit, iz; ix, ip, iz; ip	ip, ix, iz
UA_diffuse_5	k, iz, i, j	i, j, iz; i, j, k; k, iz	k, j, i, iz
UA_setup_16	i, j, ip	i, j; ip, i; ip, j; ip	i, ip, j
UA_transfer_4	col, j, i	digit, col; j, col; i, col; $i-1$, j, digit	j, i, col

Table 1. Original and permuted orders of loops

This permuted loop order is to be defined by the user. Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer. Analyzing the content of Table 1, we can conclude that in each tiled code (representing loop permutation in tiles) the last iterative variable, defining the innermost loop, is the same as the variable defining the last dimension in most arrays present in statements of a corresponding loop nest body.

The reason is the following. Programs tend to reuse data near those they have used recently (spatial locality), or that were recently referenced themselves (temporal locality). Data elements are brought into one cache line at a time and if one element

Loop Nest	Dependence Analysis	Transitive Closure Calculation	ISL Code Generation	Other Time	Total
BT_rhs_1	0.118	0.671	0.656	2.533	3.978
FT_auxfnct_2	0.001	0.001	0.004	0.768	0.774
LU_pintgr_2	0.277	0.251	1.126	0.858	2.462
SP_nivr_1	0.300	0.220	0.004	1.327	1.887
UA_adapt_10	0.030	0.021	0.557	1.575	2.183
UA_diffuse_5	0.018	0.001	0.081	0.545	0.645
UA_setup_16	0.010	0.001	0.047	0.495	0.553
UA_transfer_4	0.002	0.002	0.114	0.594	0.748

Table 2. Time of particular stages of code generation (in seconds)

is referenced, a few neighboring elements will also be brought into cache. If these neighboring elements will be referenced by successive instructions, there will be no cache miss penalty, this reduces code execution time. For the C language, such a situation takes place when both the last iterative variable defining the innermost loop and the index defining the last array dimension in a corresponding loop nest body are the same.

When we deal with multiple arrays, with some arrays accessed by rows and some by columns, we choose the iterative variable available in statements of a corresponding loop to be last in permuted tiled code that represents the last dimension in most arrays available in statements of the loop nest body.

The goal of experiments was to evaluate the time of code generation according to the introduced algorithm and compare speed-ups of original and permuted tiled codes. To carry out experiments, we have used a computer with the following features: 2 × Intel Xeon CPU E5-2695 v2, 2.40 GHz, 12 Cores, 24 Threads, 30 MB Cache, 16 GB RAM.

Source and target codes of the examined programs are available at the website [37].

Parallel code was generated automatically by TRACO with an option allowing for extraction of synchronization-free parallelism based on the algorithm presented in paper [14]. TRACO applies that algorithm to a dependence graph whose vertices are target tiles while each directed edge points out a dependence between a pair of corresponding target tiles. TRACO generates parallel code in the OpenMP standard [17].

Both original and tiled codes (with the tile side equal to 16) were compiled by means of the GCC 4.8.3 compiler. The dimension of a tile including instances of a loop nest statement is the same as the number of loops surrounding this statement.

Table 2 shows time of the particular stages of TRACO automatic code generation for an Intel i5-4670 3.4 GHz computer. Column 2 presents time of dependence analysis. Times of transitive closure calculation and code generation by means of the ISL code generator are placed in columns 3 and 4, respectively. Column 5 informs about time of other operations: dependence relations pre-processing, building sets

Loop	Permut.	CPUs →	1	2		4		8		16		32	
		l.i.u.b	T	T	S	T	S	T	S	T	S	T	S
BT_rhs_1	Yes	500	10.50	3.61	2.91	1.98	5.30	1.04	10.11	0.58	18.01	0.52	20.04
		1 000	165.13	36.04	4.58	19.25	8.58	12.47	13.25	7.35	22.47	6.07	27.22
	No	500	10.50	13.45	0.78	13.54	0.78	14.89	0.71	17.05	0.62	18.58	0.57
		1 000	165.13	96.51	1.71	189.54	0.87	175.11	0.94	206.33	0.80	159.62	1.03
FT_auxfnct_2	Yes	500	8.07	1.25	6.48	0.68	11.89	0.38	21.24	0.23	35.09	0.22	37.36
		1 000	90.40	7.71	11.73	6.16	14.68	3.78	23.94	1.99	45.43	1.61	56.08
	No	500	8.07	1.65	4.88	0.78	10.39	0.44	18.22	0.28	28.92	0.27	29.67
		1 000	90.40	12.15	7.44	8.20	11.03	5.75	15.72	2.91	31.08	2.13	42.42
LU_pintgr_2	Yes	5 000	0.61	0.35	1.76	0.15	4.21	0.12	5.08	0.06	9.53	0.06	9.84
		10 000	2.46	1.27	1.94	0.68	3.61	0.44	5.59	0.34	7.35	0.29	8.37
	No	5 000	0.61	0.79	0.77	0.79	0.77	1.57	0.39	1.95	0.31	1.06	0.57
		10 000	2.46	2.70	0.91	3.46	0.71	3.23	0.76	3.94	0.62	3.65	0.67
SP_ninvr_1	Yes	250	1.34	0.95	1.41	0.61	2.21	0.43	3.10	0.28	4.81	0.26	5.08
		500	41.47	9.06	4.58	4.34	9.56	2.44	16.97	1.65	25.21	1.28	32.35
	No	250	1.34	1.03	1.31	0.68	1.97	0.38	3.49	0.31	4.35	0.29	4.59
		250	41.47	10.12	4.10	6.29	6.60	3.70	11.22	2.23	18.56	1.59	26.08
UA_adapt_10	Yes	500	0.73	0.54	1.37	0.51	1.44	0.33	2.23	0.34	2.16	0.25	2.94
		1 000	5.16	3.20	1.61	2.81	1.83	2.57	2.01	2.47	2.09	1.99	2.60
	No	500	0.73	0.40	1.84	0.54	1.35	0.45	1.62	0.34	2.16	0.31	2.38
		1 000	5.16	2.99	1.73	3.98	1.30	2.48	2.08	2.54	2.03	1.95	2.65
UA_diffuse_5	Yes	200	7.36	2.84	2.59	2.54	2.89	2.06	3.58	1.83	4.03	1.81	4.06
		300	51.10	13.62	3.75	11.18	4.57	10.67	4.79	10.44	4.90	9.42	5.42
	No	200	7.36	2.69	2.73	3.09	2.38	2.55	2.88	2.21	3.33	2.17	3.40
		300	51.10	15.05	3.40	13.65	3.74	12.90	3.96	11.80	4.33	13.03	3.92
UA_setup_16	Yes	700	0.54	0.50	1.09	0.46	1.20	0.38	1.42	0.37	1.49	0.32	1.68
		900	4.71	1.26	3.75	0.90	5.24	0.81	5.81	0.74	6.35	0.79	5.95
	No	700	0.54	0.78	0.70	0.51	1.06	0.43	1.26	0.50	1.08	0.42	1.28
		900	4.71	1.50	3.14	1.07	4.41	1.05	4.48	0.94	5.00	0.79	5.98
UA_transfer_4	Yes	1 250	2.49	2.01	1.24	1.96	1.27	1.86	1.34	1.66	1.50	1.56	1.60
		2 500	50.42	19.85	2.54	17.78	2.84	15.02	3.36	14.16	3.56	12.15	4.15
	No	1 250	2.49	2.30	1.09	2.28	1.09	1.96	1.27	1.94	1.28	1.81	1.38
		2 500	50.42	20.42	2.47	18.08	2.79	16.49	3.06	15.76	3.20	15.20	3.32

Table 3. Time (T, in seconds) and speed-up (S); "l.i.u.b." denotes loop index upper bounds defining the problem size

Loop Nest	**Type**	**Speed-Up**	**Reason**
BT_rhs_1	perf	high	permutation increases both temporal and spatial locality; computational work per processor is enough to increase speed-up with increasing the number of CPUs up to 32
FT_auxfnct_2	perf	high	permutation increases both temporal and spatial locality; computational work per processor is enough to increase speed-up with increasing the number of CPUs up to 32
LU_pintgr_2	perf	low for the larger number of CPUs	permutation increases both temporal and spatial locality; computational work per processor is not enough to increase considerably speed-up with increasing the number of CPUs, from 8 up to 32
SP_nivr_1	perf	high	permutation increases both temporal and spatial locality; computational work per processor is enough to increase speed-up with increasing the number of CPUs up to 32
UA_adapt_10	imperf	low	permutation does not increase locality, the possible reason is that multiple inner loop nests representing tiles are generated; while execution, each such next loop nest destructs cache data formed by the previous nest.
UA_diffuse_5	perf	low for the larger number of CPUs	permutation increases temporal locality for array $r[i][j][iz]$, but does not enhance locality of array $u[i][j][k]$
UA_setup_16	perf	low for the larger number of CPUs	permutation increases both temporal and spatial locality; computational work per processor is not enough to increase considerably speed-up with increasing the number of CPUs from 8 up to 32
UA_transfer_4	imperf	low for the larger number of CPUs	permutation slightly increases locality; computational work per processor is not enough to increase considerably speed-up with increasing the number of CPUs from 8 up to 32

Table 4. Types of loop nests and the explanation of the behavior of tiled code speed-up, "perf" and "imperf" stand for perfectly and imperfectly nested loops, respectively

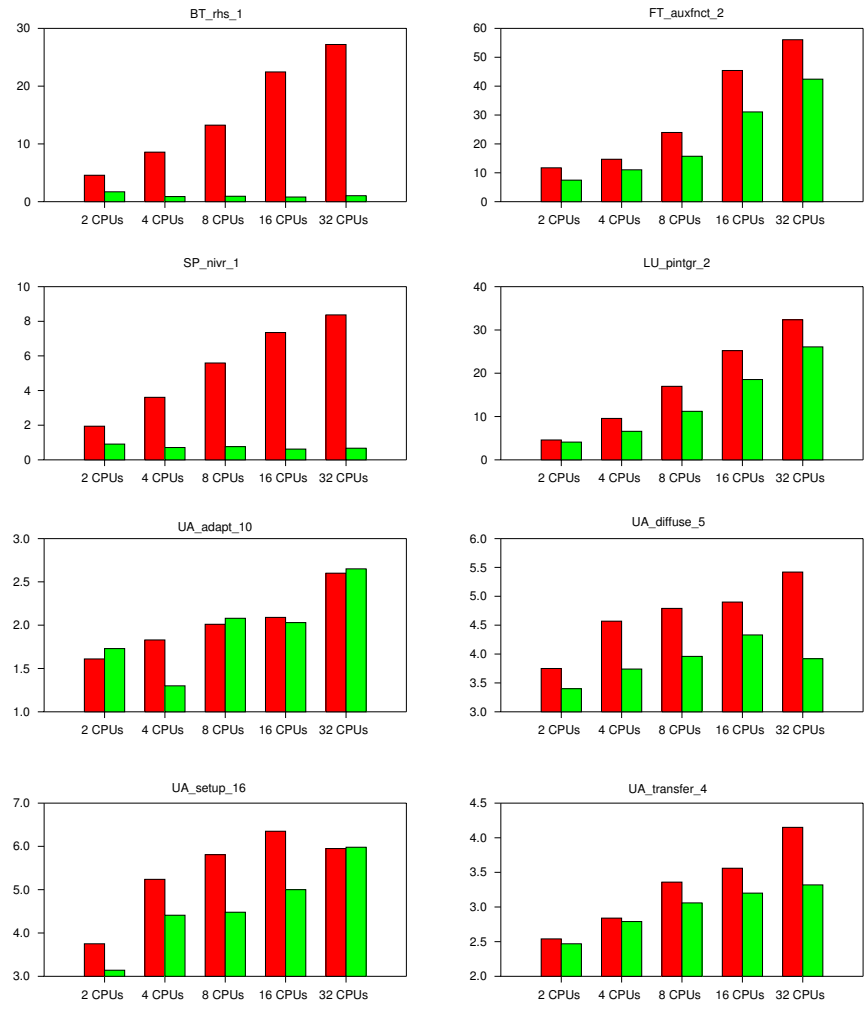

Figure 5. Speed-up of tiled original (green bars) and permuted (red bars) codes

envisaged by the introduced algorithm, sets and relations simplification, and code post-processing. The last column shows total time. As we can see from Table 2, the time of the calculation of transitive closure takes less than 15 % of the total time for each benchmark under experiments.

Table 3 presents execution time (T) in seconds and speed-up (S) for the studied loop nests when the GCC compiler was applied with the -O3 optimization option. All the values of upper loop index bounds were defined to be the same. Speed-up was calculated as the ratio of an original sequential program execution time to a corresponding tiled parallel program execution time on P processors.

Figure 5 shows speed-up of parallel tiled codes in a graphical way when each loop index upper bound is equal to the maximal value presented in Table 3.

The conclusions of qualitative analysis of speed-up received are presented in Table 4. The second column clarifies what is the type of a corresponding loop nest: perfectly or imperfectly nested. The third column shows the character of speed-up depending on the number of CPUs used. The last column, for each loop nest, presents a reason of speed-up behavior.

For all codes being examined, except from the *UA_adapt_10* program, loop permutation within tiles leads to increasing code locality and, as a consequence, to increasing parallel code speed-up. For the *UA_adapt_10* program, because the loop nest is imperfectly nested and includes multiple statements, while executing tiled code each thread scans multiple tiles for a given value of the index of the outermost loop. Scanning each next tile leads to destructing data (associated with the previous tile) in cache, this makes impossible increasing code locality due to loop permutation.

6 CONCLUSION

In this paper, we presented an extended approach (in comparison with that outlined in the paper [6]) allowing for parallel tiled code generation with permutation of loops within tiles. It is based on a combination of the Polyhedral and Iteration Space Slicing frameworks and allows for an arbitrary order of loops within tiles. This new order is defined by the user as input data. The approach produces compilable parallel tiled OpenMP C/C++ code representing a permuted order of loops within tiles. We demonstrated by means of a subset of the NPB benchmark suite that a proper order of loops within tiles improves code locality that leads to increasing speed-up of tiled code.

In the future, we plan to combine iteration space slicing [14], free scheduling [15], tiling, and permutation in one framework to allow users to manage the locality, parallelism degree, and granularity of target parallel tiled code.

REFERENCES

[1] Bondhugula, U.—Hartono, A.—Ramanujam, J.—Sadayappan, P.: A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. ACM SIGPLAN Notices – PLDI '08, Vol. 43, 2008, No. 6, pp. 101–113, doi: 10.1145/1375581.1375595.

[2] Griebl, M.: Automatic Parallelization of Loop Programs for Distributed Memory Architectures. Habilitation, University of Passau, 2004, 207 pp.

[3] Irigoin, F.—Triolet, R.: Supernode Partitioning. Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '88), ACM, 1988, pp. 319–329, doi: 10.1145/73560.73588.

[4] Lim, A.—Cheong, G. I.—Lam, M. S.: An Affine Partitioning Algorithm to Maximize Parallelism and Minimize Communication. Proceedings of the 13th Interna-

tional Conference on Supercomputing (ICS '99), ACM Press, 1999, pp. 228–237, doi: 10.1145/305138.305197.

[5] Xue, J.: Loop Tiling for Parallelism. Kluwer Academic Publishers, 2000, doi: 10.1007/978-1-4615-4337-4.

[6] Bielecki, W.—Palkowski, M.: Perfectly Nested Loop Tiling Transformations Based on the Transitive Closure of the Program Dependence Graph. Soft Computing in Computer and Information Science, Vol. 342, 2015, pp. 309–320, doi: 10.1007/978-3-319-15147-2_26.

[7] Bielecki, W.—Palkowski, M.: Tiling of Arbitrarily Nested Loops by Means of the Transitive Closure of Dependence Graphs. International Journal of Applied Mathematics and Computer Science, Vol. 26, 2016, No. 4, pp. 919–939.

[8] Banerjee, U. K.: Loop Parallelization. Kluwer Academic Publishers, 1994, doi: 10.1007/978-1-4757-5676-0.

[9] Bielecki, W.—Klimek, T.—Palkowski, M.—Beletska, A.: An Iterative Algorithm of Computing the Transitive Closure of a Union of Parameterized Affine Integer Tuple Relations. Fourth International Conference on Combinatorial Optimization and Applications (COCOA 2010). Lecture Notes in Computer Science, Vol. 6508, 2010, pp. 104–113.

[10] Kelly, W.—Pugh, W.—Rosser, E.—Shpeisman, T.: Transitive Closure of Infinite Graphs and Its Applications. International Journal of Parallel Programming, Vol. 24, 1996, No. 6, pp. 579–598.

[11] Verdoolaege, S.—Cohen, A.—Beletska, A.: Transitive Closures of Affine Integer Tuple Relations and Their Overapproximations. Proceedings of the 18th International Conference on Static Analysis (SAS 2011). Springer-Verlag, Lecture Notes in Computer Science, Vol. 6887, 2011, pp. 216–232, doi: 10.1007/978-3-642-23702-7_18.

[12] Bielcki, W.—Kraska, K.—Klimek, T.: Using Basis Dependence Distance Vectors to Calculate the Transitive Closure of Dependence Relations by Means of the Floyd-Warshall Algorithm. Journal of Combinatorial Optimization, Vol. 30, 2015, No. 2, pp. 253–275.

[13] Chen, C.: Omega+ Library. School of Computing University of Utah, 2011. Available on: http://www.cs.utah.edu/~chunchen/omega.

[14] Beletska, A.—Bielecki, W.—Cohen, A.—Palkowski, M.—Siedlecki, K.: Coarse-Grained Loop Parallelization: Iteration Space Slicing vs. Affine Transformations. Parallel Computing, Vol. 37, 2011, pp. 479–497, doi: 10.1016/j.parco.2010.12.005.

[15] Bielecki, W.—Palkowski, M.—Klimek, T.: Free Scheduling for Statement Instances of Parameterized Arbitrarily Nested Affine Loops. Parallel Computing, Vol. 38, 2012, No. 9, pp. 518–532, doi: 10.1016/j.parco.2012.06.001.

[16] NAS Benchmarks Suite, 2013. Available on: http://www.nas.nasa.gov.

[17] OpenMP Architecture Review Board, OpenMP Application Program Interface Version 4.0, 2012. Available on: http://www.openmp.org/mp-documents/OpenMP4.0RC1_final.pdf.

[18] Verdoolaege, S.: Integer Set Library – Manual. Technical report, 2011. Available on: www.kotnet.org/~skimo/isl/manual.pdf.

[19] Kelly, W.—Maslov, V.—Pugh, W.—Rosser, E.—Shpeisman, T.—Wonnacott, D.: The Omega Library Interface Guide. Technical report, College Park, MD, USA, 1995.

[20] Bondhugula, U. K. R.: Effective Automatic Parallelization and Locality Optimization Using the Polyhedral Model. Ph.D. thesis, The Ohio State University, Columbus, OH, USA, 2008.

[21] Wonnacott, D. G.—Strout, M. M.: On the Scalability of Loop Tiling Techniques. Proceedings of the 3rd International Workshop on Polyhedral Compilation Techniques (IMPACT 2013), 2013.

[22] Feautrier, P.: Some Efficient Solutions to the Affine Scheduling Problem: I. One-Dimensional Time. International Journal of Parallel Programming, Vol. 21, 1992, No. 5, pp. 313–347.

[23] Feautrier, P.: Some Efficient Solutions to the Affine Scheduling Problem: II. Multidimensional Time. International Journal of Parallel Programming, Vol. 21, 1992, No. 6, pp. 389–420.

[24] Bastoul, C.: Code Generation in the Polyhedral Model Is Easier Than You Think. IEEE International Conference on Parallel Architecture and Compilation Techniques (PACT '13), Antibes Juan-les-Pins, France, 2004, pp. 7–16, doi: 10.1109/PACT.2004.1342537.

[25] Pugh, W.—Rosser, E.: Iteration Space Slicing and Its Application to Communication Optimization. Proceedings of the 11th International Conference on Supercomputing (ICS '97), 1997, pp. 221–228, doi: 10.1145/263580.263637.

[26] The Polyhedral Benchmark Suite, 2012. Available on: `http://www.cse.ohio-state.edu/~pouchet/software/polybench/`.

[27] Verdoolaege, S.: Barvinok: User Guide. Version: Barvinok-0.36. 2012. Available on: `http://garage.kotnet.org/~skimo/barvinok/barvinok.pdf`.

[28] Lim, A. W.—Lam, M. S.: Communication-Free Parallelization via Affine Transformations. Languages and Compilers for Parallel Computing (LCPC 1994). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 892, 1994, pp. 92–106, doi: 10.1007/BFb0025873.

[29] Ramanujam, J.—Sadayappan, P.: Tiling Multidimensional Iteration Spaces for Multicomputers. Journal of Parallel and Distributed Computing, Vol. 16, 1992, No. 2, pp. 108–120, doi: 10.1016/0743-7315(92)90027-K.

[30] Wolf, M. E.—Lam, M. S.: A Data Locality Optimizing Algorithm. ACM SIGPLAN Notices, Vol. 26, 1991, No. 6, pp. 30–44.

[31] Wolf, M. E.—Lam, M. S.: A Loop Transformation Theory and an Algorithm to Maximize Parallelism. IEEE Transactions on Parallel and Distributed Systems, Vol. 2, 1991, No. 4, pp. 452–471.

[32] Krishnamoorthy, S.—Baskaran, M.—Bondhugula, U.—Ramanujam, J.—Rountev, A.—Sadayappan, P.: Effective Automatic Parallelization of Stencil Computations. ACM SIGPLAN Notices – Proceedings of the 2007 PLDI Conference, Vol. 42, 2007, No. 6, pp. 235–244, doi: 10.1145/1250734.1250761.

[33] Mullapudi, R. T.—Bondhugula, U.: Tiling for Dynamic Scheduling. Fourth International Workshop on Polyhedral Compilation Techniques (IMPACT 2014), 2014.

[34] Wonnacott, D.—Jin, T.—Lake, A.: Automatic Tiling of "Mostly-Tileable" Loop Nests. Fifth International Workshop on Polyhedral Compilation Techniques (IMPACT 2015), 2015.

[35] Bielecki, W.—Palkowski, M.—Klimek, T.: Free Scheduling of Tiles Based on the Transitive Closure of Dependence Graphs. In: Wyrzykowski, R., Deelman, E., Dongarra, J., Karczewski, K., Kitowski, J., Wiatr, K. (Eds.): Parallel Processing and Applied Mathematics. Springer, Cham, Lecture Notes in Computer Science, Vol. 9574, 2015, pp. 133–142.

[36] Palkowski, M.—Klimek, T.—Bielecki, W.: TRACO: An Automatic Loop Nest Parallelizer for Numerical Applications. Federated Conference on Computer Science and Information Systems (FedCSIS), 2015, pp. 681–686, doi: 10.15439/2015F34.

[37] Source and Target Codes, `http://sourceforge.net/p/traco/code/HEAD/tree/trunk/examples/perm/`.

Marek Palkowski graduated and obtained his Ph.D. degree in computer science from the Technical University of Szczecin, Poland. The main goal of his research is extraction of parallelism available in program loop nests using the transitive closure of dependence graphs, development of the publicly available TRACO compiler implementing parallelization techniques based on the transitive closure of dependence graphs.

Wlodzimierz Bielecki is Full Professor, Head of the Software Technology Department of the West Pomeranian University of Technology, Szczecin. His research interest includes parallel and distributed computing, optimizing compilers, techniques of extraction of both fine- and coarse-grained parallelism available in program loop nests based on the transitive closure of dependence graphs.