Computer Science • 18(2) 2017 http://dx.doi.org/10.7494/csci.2017.18.2.195

Włodzimierz Bielecki
Piotr Skotnicki

DYNAMIC TILE FREE SCHEDULING FOR CODE WITH ACYCLIC INTER-TILE DEPENDENCE GRAPHS

Abstract *Free scheduling is a task ordering technique under which instructions are executed as soon as their operands become available. Coarsening the grain of computations under the free schedule, by means of using groups of loop nest statement instances (tiles) in place of single statement instances, increases the locality of data accesses and reduces the number of synchronization events, and as a consequence improves program performance. The paper presents an approach for code generation that allows for the free schedule for tiles of arbitrarily nested affine loops at run-time. The scope of the applicability of the introduced algorithms is limited to tiled loop nests whose inter-tile dependence graphs are cycle-free. The approach is based on the polyhedral model. Results of experiments with the PolyBench benchmark suite, demonstrating significant tiled code speed-up, are discussed.*

Keywords optimizing compilers, tiling, task scheduling, parallel computing, dependence graph, data locality

Citation Computer Science 18(2) 2017: 195–216

1. Introduction

One of the goals of code transformations applied by an optimizing compiler is to reorganize loop nest iterations so that their execution is performed faster without altering program semantics. This objective is achieved either by extracting independent calculations and/or by means of grouping together statement instances accessing a common memory addressing space, improving the locality of computations as a result. On the whole, code transformations are mostly aimed at finding a new legal execution order of loop nest statement instances.

In theory, free scheduling [4] is the fastest known ordering technique under which instructions are executed – possibly in parallel – as soon as their operands become available.

In practice, however, taking into account the architecture of modern processing units – characterized by a limited number of cores and a multi-level memory hierarchy – the approach can fail to yield satisfactory speed-up. When the tasks to be scheduled are loop nest statement instances, the main drawback of free scheduling is its fine granularity which can lead to a significant number of synchronization events.

Tiling [10] is a well-known iteration reordering transformation aimed at increasing the locality of computations. The technique involves partitioning the points of an iteration space into smaller blocks (known as *tiles*) so that data chunks needed for computations performed within each block fit in a cache memory and are reused multiple times before getting evicted, hence reducing the total number of reads and writes to a shared storage.

In this paper, we present an approach for code generation that allows for the free schedule for statically formed tiles of arbitrarily nested affine loops at run-time. The scope of the applicability of the algorithms is limited to tiled loop nests whose inter-tile dependence graphs are cycle-free.

Our approach is based on the polyhedral model [5–7]. Experimental results demonstrate that the overhead induced by redundant operations, required to define tiles to be executed at run-time, can be outbalanced by a sufficient number of parallel tiles, a reduced number of synchronization events, and an increased level of data locality, all of which eventually lead to satisfactory code speed-up.

The main contributions of this paper over previous work can be summarized as follows:

- we propose an approach that allows to define first static tiles for arbitrarily nested loops and then execute them under the free scheduling at run-time;
- we demonstrate that computing the free schedule does not require neither any affine transformation nor dependence graph transitive closure; as a consequence, this leads to the low time-complexity of this approach and allows to obtain satisfactory parallel tiled code speed-up;
- we describe a simple way to detect whether an inter-tile dependence graph is cycle-free and demonstrate its effectiveness for the PolyBench loop nests;

- we present publicly available academic software, the TC compiler, implementing the presented approach;
- we discuss the speed-up of parallel tiled code generated by means of the presented approach.

The rest of the paper is organized as follows. The next section introduces the background and mathematical theory. Section 3 presents the algorithm and demonstrates its application to a working loop nest. The results of our experiments are discussed in Section 4. Section 5 revises related techniques. In the summary, we conclude and present our plans for future research.

2. Background

In this paper, we deal with affine loop nests where lower and upper bounds, array subscripts, and conditionals are affine functions of surrounding loop indices and symbolic constants, and the loop steps are known constants.

The presented algorithm is based on the polyhedral model [5–7]. 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 providing a data dependence analysis based on this representation; ii) program transformation, with the aim of improving program locality and/or parallelization; iii) code generation.

A loop nest is *perfectly nested* if all of its statements are surrounded by the same loops; otherwise, the loop nest is *imperfectly nested*.

A *statement instance* $\mathrm{S}[I]$ is a particular execution of loop statement S for given iteration I. Two statement instances $\mathrm{S1}[I]$ and $\mathrm{S2}[J]$ are *dependent* if both access the same memory location and at least one access is a write. $\mathrm{S1}[I]$ and $\mathrm{S2}[J]$ are called the source and the target of a dependence, respectively, provided that $\mathrm{S1}[I]$ is executed before $\mathrm{S2}[J]$. The sequential ordering of statement instances, denoted $\mathrm{S1}[I] \prec \mathrm{S2}[J]$, is induced by the original execution ordering of iteration vectors, or by the textual ordering of statements if $I = J$.

An iteration vector can be represented by a k-integer tuple of loop indices in the $\mathbb{Z}^k$ iteration space. Consequently, a dependence relation is a mapping from tuples to tuples of the form $\{\,[source] \to [target] \mid constraints\,\}$, where *source* defines dependence sources, *target* defines dependence targets, and *constraints* is a *Presburger formula* (built of affine equalities and inequalities, logical and existential operators) that imposes constraints on the variables and/or expressions within *source* and *target* tuples.

It is often convenient to group related elements of a set – like instances of the same statement in particular – using a *named integer tuple* [21]. Typically, a name associated with a tuple is the same as the label of a corresponding statement (if the tuple represents an iteration vector), or as the name of a variable (if the tuple represents memory offsets in subsequent array dimensions).

Let LD denote a loop nest iteration domain comprising all statement instances. A *schedule* is a function $\sigma : LD \rightarrow \mathbb{Z}$ that assigns a discrete time of execution (*timestamp*) to each loop nest statement instance.

A schedule is *valid* if for each pair of dependent statement instances, $\mathrm{S1}[I]$ and $\mathrm{S2}[J]$, satisfying the condition $\mathrm{S1}[I] \prec \mathrm{S2}[J]$, the condition $\sigma(\mathrm{S1}[I]) < \sigma(\mathrm{S2}[J])$ holds true, i.e., the dependences are preserved when statement instances are executed in an increasing order of their timestamps. If $\sigma(\mathrm{S1}[I]) = \sigma(\mathrm{S2}[J])$, statements $\mathrm{S1}[I]$ and $\mathrm{S2}[J]$ can be computed in parallel.

The *free schedule* (1) is the schedule that assigns the earliest valid timestamp to each loop statement instance, i.e., as soon as the statement dependences are resolved [4],

$$\sigma(p) = \begin{cases} 0; \text{ if there is no } p' \in LD\, s.t.\, p' \rightarrow p \in R, \\ 1 + \max(\sigma(p_1), \sigma(p_2), ..., \sigma(p_n)); \; p, p_1, p_2, ..., p_n \in LD \, \wedge \\ \qquad p_1 \rightarrow p, p_2 \rightarrow p, ..., p_n \rightarrow p \in R, \end{cases} \tag{1}$$

where:

$p, p', p_1, p_2, ..., p_n$ are statement instances,
R is a relation describing data dependences,
n is the number of dependences whose destination is statement instance p.

An *access relation* is a mapping $\varphi : LD \rightarrow D$, where D is a data space, that associates a statement instance with memory locations that are read from (*read access relation*) or written to (*write access relation*) by the source statement instance.

For manipulating sets and relations, we use common operations, such as intersection ($\cap$), union ($\cup$), difference ($-$), composition ($\circ$), domain of a relation (domain(R)), range of a relation (range(R)), application of a relation to a set ($R(S)$), restriction of the domain of a relation ($\backslash$).

3. Tiling algorithm

In this section, we present a working example, static calculations, and code generation allowing for tile execution under the free schedule at run-time.

The key idea of the approach introduced in this paper is to split a tiling procedure into two parts. First, compile-time computations translate the input code into its polyhedral representation, group iteration points into rectangular tiles and form a (possibly infinite) directed graph representing inter-tile dependences; code scanning statement instances contained within a single parametric tile is generated. Then, the graph is checked to determine whether it is cycle-free; if so, code is generated to form the tile free schedule at run-time and to execute statements associated with tiles at each timestamp.

3.1. Static computations

Given a loop nest of q arbitrarily nested statements, the algorithm starts with extracting a polyhedral model from the input code to yield the following data: loop nest iteration domain (LD_i) of each statement S$i, i = 1, ..., q$, schedule (S), and read/write access relations (RA, WA, respectively). The loop nest iteration domain is the set of statement instances executed by a loop nest for each statement. 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. An access relation maps an iteration vector to one or more memory locations of array elements.

Next, we apply a rectangular tiling scheme to each statement. For each statement S$i, i = 1, ..., q$, surrounded by d_i loops, a corresponding tile can be represented by parametric set $TILE_i, i = 1, ..., q$, that groups statement instances included in a block identified by given symbolic constants (2),

$$\begin{aligned} TILE_i(II_i) = & [II_i] \to \{ [I_i] \mid B_i * II_i + \\ & LB_i \le I_i \le \min(B_i * (II_i + 1_i) + LB_i - 1_i, UB_i) \wedge II_i \ge 0_i \} \end{aligned} \tag{2}$$

where vectors LB_i and UB_i include the lower and upper bounds, respectively, of the indices of loops enclosing statement Si; diagonal matrix B_i defines the size of a rectangular original tile; elements of vectors I_i and II_i represent the original indices of loops enclosing statement Si and the identifiers of tiles, respectively; 1_i and 0_i are the vectors whose all d_i elements have value 1 and 0, respectively.

Additionally, with each set $TILE_i, i = 1, ..., q$, we associate another set, $II\_SET_i, i = 1, ..., q$ (3), that includes the identifiers of all the tiles in a corresponding iteration space.

$$II\_SET_i = \{ [II_i] \mid II_i \ge 0_i \wedge B_i * II_i + LB_i \le UB_i \} . \tag{3}$$

The introduced approach requires an exact representation of loop-carried and loop-independent data dependences. The relation representing dependences can be computed according to formula (4) presented in paper [21].

$$R = ((RA^{-1} \circ WA) \cup (WA^{-1} \circ RA) \cup (WA^{-1} \circ WA)) \cap (S \prec S) . \tag{4}$$

Formula (4) calculates a union of flow, anti, and output dependences that exist under the lexicographic order of timestamps of all statement instances. $S \prec S$ denotes a strict partial order of statement instances, computed as: $S^{-1} \circ (\{ [e] \to [e'] \mid e \prec e' \} \circ S)$.

We aim at tiling both perfectly and imperfectly nested loops. This implies that the computations on sets and relations need to be fulfilled regardless of which iteration space a particular statement instance belongs to. While this may not be an issue for perfectly nested loops – whose all statements reside in a single iteration space – it gives rise to the problem of applying operations to sets and relations with different sizes of tuples in the case of arbitrarily nested loops; i.e., a dependence source can belong to one iteration space while the corresponding dependence target can reside in a distinct space, for example, $S1[1] \to S2[2,4]$. To make computations on sets and relations feasible, and enable the handling of separate statements uniformly, all of the sets and relations that are subject to further processing need to be *normalized.* A normalization procedure involves extending corresponding tuples with extra dimensions that will i) make tuple sizes to be equal, ii) allow us to unambiguously identify which statement a given normalized tuple refers to.

To normalize sets and relations, we apply the following transformation: given a loop nest of maximal depth d, we extend the tuples of these sets and relations to length $2d$. The elements of each tuple are built from a series of d pairs. Each pair corresponds to a single loop. The first element of such a pair holds the value of the iterator of its associated loop; the second value is the numerical order of the statement relative to a directly enclosing loop (a loop with its body is considered as an indivisible statement at a given depth). Pairs within each tuple are ordered starting from the outermost to the innermost loop, each enclosing the associated statement. The remaining elements of tuples whose statements are enclosed by fewer than d loops are filled with the value 0. Eventually, from each tuple we remove each element corresponding to a numerical order that has the value 0 in all normalized tuples. If the analyzed static control part includes more statements at depth 0, we insert an additional element at the leftmost position of each tuple, indicating the numerical order of the corresponding outermost statement relative to that static control part.

In practice, to make sets and/or relations to be normalized, we can apply a scheduling function S computed by the Polyhedral Extraction Tool [22] to tuples of these sets and/or relations. We will denote the union of all normalized sets $TILE_i$ and II_SET_i, $i = 1, ..., q$, as $TILE$ and II_SET, respectively.

Next, we are interested in constructing a relation, R_TILE, describing inter-tile dependences: a dependence between tiles exists iff there exists a data dependence such that its source originates from a statement instance within one tile and targets a statement instance included in another tile. For this purpose, we adapt the idea presented in paper [14] to form relation R_TILE (5),

$$\begin{aligned} R\_TILE = \{\, [II] \to [JJ] \mid II, JJ \in II\_SET \wedge II \neq JJ \wedge \\ \exists I, J : I \in TILE(II) \wedge J \in TILE(JJ) \wedge J \in R(I) \,\}, \end{aligned} \tag{5}$$

where II, JJ are vectors representing tile identifiers, while I, J are vectors representing statement instances. This relation ignores intra-tile dependences – they will be respected by sequential execution of statement instances within each tile.

It is well-known that for a cycle-free graph, a legal schedule for the vertices of this graph can be found [5, 6]. However, because we form rectangular tiles that can be of arbitrary sizes without considering possible inter-tile dependences, we do not guarantee that a corresponding dependence graph will be cycle-free. We would therefore like to ensure that a graph is acyclic. With this purpose, we check whether relation R_TILE is lexicographically forward [11], i.e., iff $\forall x \to y \in A, y - x \succ 0$ ($y - x$ is lexicographically positive). If so, this guarantees that a graph is cycle-free [11] and obviates the need to compute the positive transitive closure of relation R_TILE (which may be expensive). In the case of the presence of lexicographically backward edges, we conclude that the graph may contain cycles, therefore, the algorithm cannot be applied. Despite the fact that the presented technique is very simple, we demonstrate in Section 4 that it is able to recognize all of the PolyBench benchmarks whose corresponding inter-tile dependence graphs are cyclic. Techniques aimed at eliminating cycles are discussed in Section 5.

Algorithm 1 provides a formal description of the approach discussed in this subsection. Step 1 transforms a loop nest into its polyhedral representation. Steps 2–7 form sets and relations that are subject to run-time computation of the free schedule. Step 8 builds an inter-tile relation. Steps 9–11 verify whether the inter-tile relation is cycle-free – if not, dynamic scheduling cannot be applied. Step 12 generates code for the execution of statement instances included in a single parametric tile.

```
Algorithm 1: Static calculations for the purpose of dynamic tiling.
  Input: Arbitrarily nested affine loops.
  Output: Relation $R\_TILE$, set $II\_SET$, code scanning the elements of set
          $TILE$.
    1. Transform the loop nest into its polyhedral representation including:
       an iteration space, access relations, and global schedule $S$.
    2. For each $i$, $i = 1, 2, ..., q$, and $d_i$, where $q$ is the number of loop nest
       statements, and $d_i$ is the number of loops surrounding statement S$i$,
       form the following vectors, matrix and sets:
         • vector $I_i$ whose elements are original loop indices $i_1,i_2,...,i_{d_i}$;
         • vector $II_i$ whose elements $ii_1,ii_2,...,ii_{d_i}$ define the identifier of
           a tile;
         • vectors $LB_i$ and $UB_i$ whose elements are lower $lb_1,...,lb_{d_i}$ and
           upper $ub_1,...,ub_{d_i}$ bounds of indices $i_1,i_2,...,i_{d_i}$ of the enclosing
           loops, respectively;
         • vector $1_i$ whose all $d_i$ elements are equal to the value 1;
         • vector $0_i$ whose all $d_i$ elements are equal to the value 0;
         • diagonal matrix $B_i$ whose diagonal elements are constants
           $b_1,b_2,...,b_{d_i}$ defining a single tile size;
         • set $TILE_i$ including the iterations belonging to a parametric tile
           defined with parameters $ii_1,ii_2,...,ii_{d_i}$ as follows:
           $TILE_i(II_i) = [II_i] \to \{ [I_i] \mid B_i * II_i + LB_i \le I_i \le$
           $\min(B_i * (II_i + 1_i) + LB_i - 1_i, UB_i) \wedge II_i \ge 0_i \}$;
         • set $II\_SET_i$ including the identifiers of corresponding tiles:
           $II\_SET_i = \{ [II_i] \mid II_i \ge 0_i \wedge B_i * II_i + LB_i \le UB_i \}$.
```

```
Algorithm 1: cont'd
 3. Carry out a dependence analysis to produce a set of relations
    $R_{i,j}, i, j = 1, 2, ..., q$ describing all of the dependences present in this loop
    nest.
 4. Normalize the tuples of relations $R_{i,j}$ and sets $TILE_i$, $i, j = 1, 2, ..., q$ by
    applying schedule $S$, received in step 1. Introduce a new dimension in
    the normalized parameter space for each fixed element of the tuples,
    equal to the value of that element, and group the identifiers in
    sets $II\_SET_i$.
 5. Calculate a union of all normalized sets $TILE_i, i = 1, 2, ..., q$ and denote
    the result as $TILE$.
 6. Calculate a union of all normalized sets $II\_SET_i, i = 1, 2, ..., q$ and denote
    the result as $II\_SET$.
 7. Calculate a union of all normalized relations $R_{i,j}, i, j = 1, 2, ..., q$ and
    denote the result as $R$.
 8. Form a relation $R\_TILE$ representing inter-tile dependences as follows:
    $R\_TILE = \{ [II] \to [JJ] \mid II, JJ \in II\_SET \land II \neq JJ \land \exists I, J : I \in TILE(II) \land J \in TILE(JJ) \land J \in R(I) \}$.
 9. Introduce a parametric point, $P$, in the $n$-dimensional normalized space:
    $P = [ii_1, ii_2, ..., ii_n] \to \{ [i_1, i_2, ..., i_n] \mid ii_1 = i_1 \land ii_2 = i_2 \land ... \land ii_n = i_n \}$.
10. Introduce a parametric set, $P\_LT$:
    $P\_LT = [ii_1, ii_2, ..., ii_n] \to \{ [i_1, i_2, ..., i_n] \mid (i_1, i_2, ..., i_n) \prec (ii_1, ii_2, ..., ii_n) \}$.
11. Verify whether relation $R\_TILE$ is lexicographically forward, i.e., whether
    the following condition is satisfied:
    $R\_TILE(P) \cap P\_LT = \varnothing$.
    If it is not satisfied, then end, the algorithm cannot be applied.
12. Generate code enumerating statement instances of a single parametric tile,
    represented with set $TILE$, in the lexicographic order, by means of
    applying any code generator.
```

3.2. Working example

In order to illustrate the presented approach, let us consider the following working example of a multi-statement, imperfectly-structured loop nest.

Example 1.

```
    for (i = 1; i <= 4; ++i) {
S1:   B[i] = A[i+1][4] + B[i+1];
      for (j = 1; j <= 4; ++j) {
S2:     A[i][j] = A[i-1][j];
      }
    }
```

The loop nest can be translated into the following polyhedral model:

$$\bigcup_{k=1}^{2} LD_k := \{\, \mathrm{S1}[i] \mid 1 \le i \le 4 \,\} \cup \{\, \mathrm{S2}[i,j] \mid 1 \le i, j \le 4 \,\},$$

$$\begin{aligned} S &:= \{\, \mathrm{S1}[i] \to [i,0,0] \,\} \cup \{\, \mathrm{S2}[i,j] \to [i,1,j] \,\}, \\ RA &:= \{\, \mathrm{S2}[i,j] \to \mathrm{A}[-1+i,j] \mid 1 \le i,j \le 4 \,\} \cup \\ &\quad\ \{\, \mathrm{S1}[i] \to \mathrm{A}[1+i,4] \mid 1 \le i \le 4 \,\} \cup \\ &\quad\ \{\, \mathrm{S1}[i] \to \mathrm{B}[1+i] \mid 1 \le i \le 4 \,\}, \\ WA &:= \{\, \mathrm{S2}[i,j] \to \mathrm{A}[i,j] \mid 1 \le i,j \le 4 \,\} \cup \{\, \mathrm{S1}[i] \to \mathrm{B}[i] \mid 1 \le i \le 4 \,\}. \end{aligned}$$

A data dependence analysis (4) over the access relations reveals the following data dependences:

$$\begin{aligned} \bigcup_{k=1}^{2}\bigcup_{l=1}^{2} R_{k,l} := &\{\, \mathrm{S1}[i] \to \mathrm{S1}[1+i] \mid 1 \le i \le 3 \,\} \cup \\ &\{\, \mathrm{S2}[i,j] \to \mathrm{S2}[1+i,j] \mid 1 \le i \le 3 \wedge 1 \le j \le 4 \,\} \cup \\ &\{\, \mathrm{S1}[i] \to \mathrm{S2}[1+i,4] \mid 1 \le i \le 3 \,\}. \end{aligned}$$

The data dependences described by relation R can be presented by means of a directed graph whose edges connect pairs of dependent statement instances represented by vertices. In particular, there exists an edge between two vertices iff one defines the source of a dependence and another defines the target of this dependence. As far as the working example is considered, the corresponding dependence graph is shown in Figure 1a. It is worth to note that we deal with the two separate iteration spaces, one for each statement – the labels `i(S2)` and `j(S2)` denote the 2-D iteration space of statement S2, and correspond to iterators i and j, respectively, i.e., for each statement instance $\mathrm{S2}[i,j]$ there exists a corresponding iteration point in that space; the label `i(S1)` denotes the 1-D iteration space of iterator i for statement S1.

For the purpose of our demonstration, we use tiles of the size 2 for S1, and of the size 2×2 for S2. Based on the working loop nest iteration domain and the tile sizes, we define the following two parametric sets (2):

$$\begin{aligned} TILE_1 := [ii] \to \{\, \mathrm{S1}[i] \mid\ & i \ge 1 + 2ii \wedge ii \le 1 \wedge i \le 2 + 2ii \wedge \\ & ii \ge 0 \wedge i \ge 1 \wedge i \le 4 \,\}, \end{aligned}$$

$$\begin{aligned} TILE_2 := [ii, jj] \to \{\, \mathrm{S2}[i,j] \mid\ & i \ge 1 + 2ii \wedge ii \le 1 \wedge i \le 2 + 2ii \wedge \\ & ii \ge 0 \wedge j \ge 1 + 2jj \wedge jj \le 1 \wedge j \le 2 + 2jj \wedge \\ & jj \ge 0 \wedge i \ge 1 \wedge i \le 4 \wedge j \ge 1 \wedge j \le 4 \,\}, \end{aligned}$$

where ii, jj are parameters defining tile identifiers and the notation $[x, y, z, ...] \to \{\, [...] \mid \textit{constraints} \,\}$ means that $[x, y, z, ...]$ are parametric variables in the constraints of a set. For example, set $TILE_2$ for the vector $(ii = 0, jj = 1)^{\mathrm{T}}$ contains statement instances included in the set $\{\,\mathrm{S2}[1,3];\ \mathrm{S2}[1,4];\ \mathrm{S2}[2,3];\ \mathrm{S2}[2,4]\,\}$. The tiled iteration spaces for the working example are presented in Figure 1b.

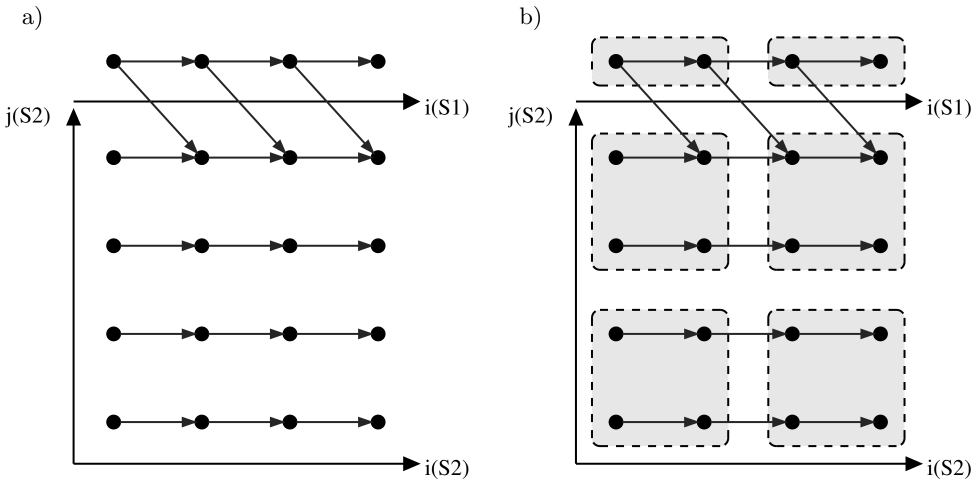

Figure 1. Dependences and tiles for the working example: a) data dependence graph; b) tiled loop nest space at time 0.

Applying formula (3) we obtain the following sets $II\_SET_i, i = 1,2$, of valid tile identifiers:

$$II\_SET_1 := \{\,[ii] \mid 0 \le ii \le 1\,\},$$

$$II\_SET_2 := \{\,[ii, jj] \mid 0 \le ii \le 1 \wedge 0 \le jj \le 1\,\}.$$

Subsequently, we normalize the computed sets and relations by means of applying the scheduling function S, which yields the normalized structures shown below:

$$\begin{aligned} TILE := [ii, kk, jj] \to \{\,[i,1,j] \mid\ & kk = 1 \wedge jj \le 1 \wedge i \ge 1 + 2ii \wedge i \le 4 \wedge i \ge 1 \wedge \\ & i \le 2 + 2ii \wedge j \ge 1 + 2jj \wedge j \ge 1 \wedge j \le 2 + 2jj\,\} \cup \\ & \{\,[i,0,0] \mid kk = 0 \wedge jj = 0 \wedge i \ge 1 + 2ii \wedge i \le 4 \wedge i \ge 1 \wedge i \le 2 + 2ii\,\}, \end{aligned}$$

$$\begin{aligned} II\_SET := & \{\,[i,1,j] \mid i \le 1 \wedge i \ge 0 \wedge j \le 1 \wedge j \ge 0\,\} \cup \\ & \{\,[i,0,0] \mid i \le 1 \wedge i \ge 0\,\}, \end{aligned}$$

$$\begin{aligned} R := & \{\,[i,1,j] \to [1+i,1,j] \mid i \le 3 \wedge i \ge 1 \wedge j \le 4 \wedge j \ge 1\,\} \cup \\ & \{\,[i,0,0] \to [1+i,1,4] \mid i \le 3 \wedge i \ge 1\,\} \cup \\ & \{\,[i,0,0] \to [1+i,0,0] \mid i \le 3 \wedge i \ge 1\,\}. \end{aligned}$$

As far as the working example is considered, relation R_TILE (5), describing inter-tile dependences, is represented as follows:

$$\begin{aligned} R\_TILE := \{\, [ii, kk, jj] \to [ii', kk', jj'] \mid\ & jj' \leq 1 - kk + jj \,\wedge \\ & jj' \geq -kk + jj + kk' \wedge jj \geq 0 \wedge jj \leq kk \wedge ii \geq 0 \,\wedge \\ & ii' \geq ii \wedge ii' \leq 1 \wedge jj' \leq jj + kk' \wedge kk' \geq kk \,\wedge \\ & jj' \geq ii + jj \wedge jj' \geq 1 + jj - ii' \,\}\,. \end{aligned}$$

The input and output tuples of relation R_TILE denote normalized identifiers of tiles. Figure 2 presents the cycle-free inter-tile dependence graph for the working example. The vertices of the graph represent single tiles and correspond to the tiles visible in Figure 1b; the edges between the vertices visualize inter-tile dependences described by R_TILE.

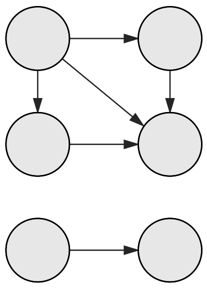

Figure 2. Inter-tile dependence graph for the working example.

3.3. Generation of code executing tiles under the free schedule

In this subsection, we show that a given relation R_TILE is enough to form the tile free schedule at run-time. We exploit the following definition of an ultimate dependence source:

Definition 1. (*Ultimate Dependence Source*) An ultimate dependence source is a source that is not a destination of another dependence. Given a dependence relation R, the set S_{UDS}, including all ultimate dependence sources, can be calculated as (6).

$$S_{UDS} = \text{domain}(R) - \text{range}(R)\,. \tag{6}$$

According to the definition of the free schedule, set S_{UDS} computed over relation R_TILE includes the tiles that have the timestamp equal to 0, i.e., they have to be executed first. Because all these tiles are independent, they can be executed in parallel. Once the computation of all these tiles is finished and all threads are synchronized, we no longer need to consider the dependences that originate from those tiles. We remove the associated vertices (along with their outgoing edges) from the dependence graph, and carry out the above procedure again. From a mathematical perspective, we have to subtract the elements of set S_{UDS} from both – the domain of relation R_TILE and from set II_SET. Figure 3a shows the modified dependence graph after applying the aforementioned steps. A newly-computed set S_{UDS} will now comprise tile identifiers that the free schedule assigns the timestamp equal to 1.

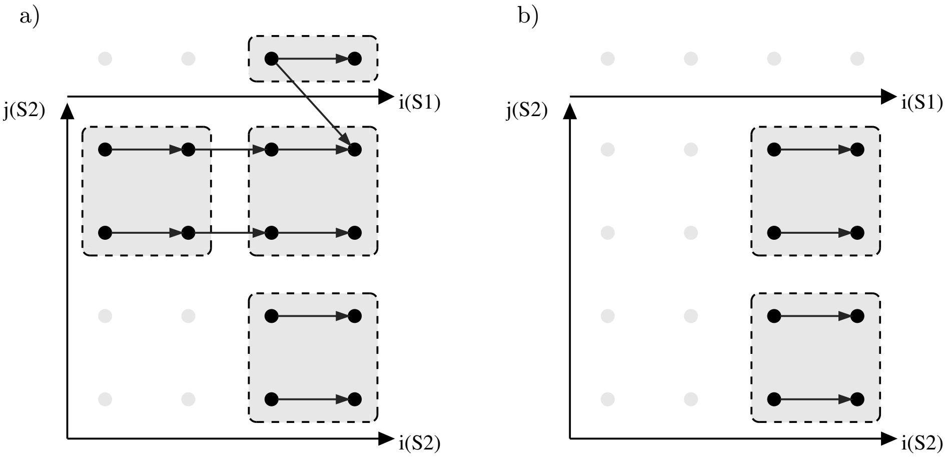

Figure 3. Tiles under the free schedule for the working example: a) remaining tiles at time 1; b) remaining tiles at time 2.

The presented procedure is repeated by means of a `while` loop at run-time, until the iteratively recomputed set S_{UDS} is empty. Tiles associated with ultimate dependence sources can be executed in parallel for a given `while` loop iteration. Parallel flows of execution must be synchronized before subsequent calculations associated with the next timestamp can be done.

Let us note that when the execution of the `while` loop finishes, there may still be statement instances in the entire loop nest iteration domain that have not been processed yet (see Figure 3b) because their associated tiles do not constitute a source of any inter-tile dependence. For this reason, each iteration of the `while` loop updates set II_SET by subtracting the identifiers of the already processed tiles, so that the remaining elements of a final set II_SET can be scheduled for parallel execution at the very end. To emit serial code enumerating statement instances within a single parametric tile, we can apply any code generator that is able to generate loops enumerating the elements of set $TILE$ in the lexicographical order. By exploiting the code generation facilities of the Integer Set Library [19], we obtain the following code for the working example.

```
if (kk == 1 && jj >= 0 && jj <= 1) {
  for (int c0 = max(1, 2 * ii + 1); c0 <= min(4, 2 * ii + 2); c0 += 1)
    for (int c2 = 2 * jj + 1; c2 <= 2 * jj + 2; c2 += 1)
      A[c0][c2] = A[c0-1][c2];
} else if (kk == 0 && jj == 0)
  for (int c0 = max(1, 2 * ii + 1); c0 <= min(4, 2 * ii + 2); c0 += 1)
    B[c0] = A[c0+1][4] + B[c0+1];
```

The statically generated code is common for all tiles. The distinction of which statement is supposed to be executed is made by the additional `if` statements, branching out the execution flow depending on the values of symbolic constants ii, kk, jj. The code is invoked as many times as there are tile identifiers included in set II_SET. The values of symbolic constants are extracted from this set in the order determined by the run-time-computed free schedule.

Algorithm 2 lists the steps of code generation that allow for executing tiles at run-time under the free schedule. One possible implementation of Algorithm 2 is presented in the next section.

Algorithm 2: Code generation for dynamic tile execution under the free schedule.

```
Input: Set $II\_SET$ comprising all tile identifiers, relation $R\_TILE$
       describing inter-tile dependences, set $TILE$ describing tiles, original
       schedule $S$.
Output: Code for dynamic free schedule-based execution of tiles.
Emit code that:

(1) fixes the values of symbolic constants of set $II\_SET$ and relation
 $R\_TILE$ with actual values of loop nest parameters

(2) computes a set of ultimate dependence sources according to the formula
$S_{UDS} \leftarrow \text{domain}(R\_TILE) - \text{range}(R\_TILE)$

(3) includes the following while loop

while $S_{UDS} \neq \varnothing$ do
    parallel for each $II \in S_{UDS}$ do
        for each $I \in TILE(II)$ under schedule $S$ do
            execute($I$)
        end
    end
    (4) modifies the sets and relation below
    $II\_SET \leftarrow II\_SET - S_{UDS}$
    $R\_TILE \leftarrow R\_TILE \setminus II\_SET$
    $S_{UDS} \leftarrow \text{domain}(R\_TILE) - \text{range}(R\_TILE)$
end
(5) executes the remaining tiles to be run at the last schedule time
parallel for each $II \in II\_SET$ do
    for each $I \in TILE(II)$ under schedule $S$ do
        execute($I$)
    end
end
```

4. Implementation and experimental study

The approach presented in this paper has been incorporated into the *TC* optimizing compiler[1] which utilizes the Polyhedral Extraction Tool [22] for extracting a polyhedral representation of an input sequence of loops and the Integer Set Library [19] for performing a dependence analysis, manipulating integer sets and relations, and generating target code.

For experiments, we have chosen the PolyBench/C 4.1 [18] benchmark suite comprising a total of 30 programs, including linear algebra kernels, data mining algorithms, stencil computations, and dynamic-programming-based solvers.

The scope of the applicability of the discussed approach is limited to acyclic inter-tile dependence graphs. Algorithm 1 finds 13 kernels in PolyBench for which an inter-tile dependence graph is cycle-free, i.e., 43% kernels in PolyBench can be tiled by means of the presented approach. The list of names of these kernels is as follows: *2mm, 3mm, atax, bicg, correlation, covariance, gemm, gemver, gesummv, mvt, syr2k, syrk, trmm.* It is worth to note that the technique for recognizing cycle-free graphs used in Algorithm 1 finds all cycle-free inter-tile dependence graphs associated with PolyBench benchmarks.

The TC compiler to generate code, executing tiles at run-time, uses the API of the Integer Set Library [19, 20] by embedding the textual representations of relation R_TILE and set II_SET directly in source code, and operating on them using functions that the library offers; i.e., we used the `isl_set_foreach_point` function for scanning the elements of set S_{UDS} as well as other common operations, including the `isl_set_subtract` and `isl_map_restrict_domain` functions. The distribution of work among threads is accomplished by means of OpenMP API [13]. In particular, we have utilized the `task` construct of the OpenMP 3.0 specification to delegate tile execution to an available thread, and the `taskwait` construct to synchronize parallel execution flows. Below, we present a reference implementation in C of our dynamic scheduler. The `create_task` function extracts the values of symbolic constants from a point-type argument and executes the statically generated code of a single tile in an OpenMP `task` region.

```
rtile = isl_map_read_from_str(ctx, /**/);
ii_set = isl_set_read_from_str(ctx, /**/);
uds = isl_set_subtract(isl_map_domain(isl_map_copy(rtile))
                     , isl_map_range(isl_map_copy(rtile)));

#pragma omp parallel
#pragma omp single
{
  while (!isl_set_is_empty(uds)) {
    isl_set_foreach_point(uds, &create_task, NULL);
    #pragma omp taskwait
```

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

```
      ii_set = isl_set_subtract(ii_set, uds);
      rtile = isl_map_intersect_domain(rtile, isl_set_copy(ii_set));
      uds = isl_set_subtract(isl_map_domain(isl_map_copy(rtile))
                            , isl_map_range(isl_map_copy(rtile)));
    }

    isl_set_foreach_point(ii_set, &create_task, NULL);
    #pragma omp taskwait
  }
```

After incorporating generated code back into a corresponding source file, the entire test suite was compiled with the GNU Compiler Collection 4.8.3 using the −O3 optimization. The result of each program execution – the content of an array that a specific kernel computes – was subsequently compared with the values produced by a corresponding original kernel.

All experiments were carried out on a multi-core, highly parallel architecture. The hardware and software configurations used to carry out the experiments are shown in Table 1.

Table 1

Environment used for experiments

Processor	Intel Xeon E5-2699 v3
Clock	2.3 GHz
Number of sockets	2
Number of cores / socket	18
Number of threads / socket	36
L1 data cache / core	32 KB
L2 cache / core	256 KB
L3 cache / socket	45 MB
RAM memory	256 GB @ 2133 MHz
Linux kernel	3.10.0 x86_64
Compiler	gcc 4.8.3
Compiler flags	–O3 –fopenmp

Under experiments, each test was repeated multiple times, using 4, 8, 16, 32, 48 and 64 threads in subsequent runs. Speed-up was computed over the serial execution time of an untransformed original kernel.

Table 2 presents the problem sizes used for the experiments, as well as the size of a tile side in each dimension, and the serial execution time of the original kernel code. For most tests, we used problem sizes classified by PolyBench as extra large data sets. For some kernels, we needed to use larger values for loop upper bounds than those defined in benchmarks, in order to divide their iteration spaces into the sufficient amount of tiles that would allow for positive speed-up. The presented size of a tile is the one that performed best out of all tested configurations and indicates the value applied to each dimension of an iteration vector. For each loop nest, the size of a single tile needed to be adjusted individually, based on a set of trials. Additionally, in order to reach positive speed-up, tile sizes needed to be increased to values ranging from 128 up to 1024. The greater the tile size is, the fewer synchronization events and run-time schedule computations are required; however, this also results in a lower parallelism degree and data locality. As a consequence, most of the tested programs reach their peak performance using fewer than 64 threads, thus leaving some of available processing units idle. That is, the size of a single tile determines the overall number of tiles in an iteration space, and the amount of parallel work in subsequent timestamps of free-schedule based processing – greater tile sizes reduce the number of tiles that can be executed in parallel at a given timestamp. At some point, the cost of spawning and synchronization of additional threads becomes large. This results in the slowdown of parallel code speed-up.

Table 2
Problem sizes used for experiments; the size of a single tile

Kernel	Problem size	Tile size	Serial time [s]
2mm	NI = 1600, NJ = 1800, NK = 2200, NL = 2400	256	11.6747
3mm	NI = 1600, NJ = 1800, NK = 2000, NL = 2200, NM = 2400	256	15.0808
atax	M = 15000, N = 15000	1024	0.3646
bicg	M = 15000, N = 15000	1024	0.3079
correlation	M = 2600, N = 3000	128	48.4425
covariance	M = 2600, N = 3000	128	48.3946
gemm	NI = 2000, NJ = 2300, NK = 2600	256	8.1275
gemver	N = 15000	512	1.1568
gesummv	N = 10000	512	0.1849
mvt	N = 15000	512	0.8037
syr2k	M = 2000, N = 2600	128	98.3569
syrk	M = 2000, N = 2600	128	9.9211
trmm	M = 2000, N = 2600	256	15.9131

Figures 4 and 5 summarize speed-up and efficiency achieved for each examined kernel. As mentioned above, the large tile sizes used in our experiments do not allow to achieve considerable parallel code performance, but the limited cost of dynamic scheduling overheads leads to visible speed-up. In the case of relatively simple loop nests like *gesummv* whose serial execution time is low, the dynamic schedule overheads result in the slowdown of parallel program speed-up.

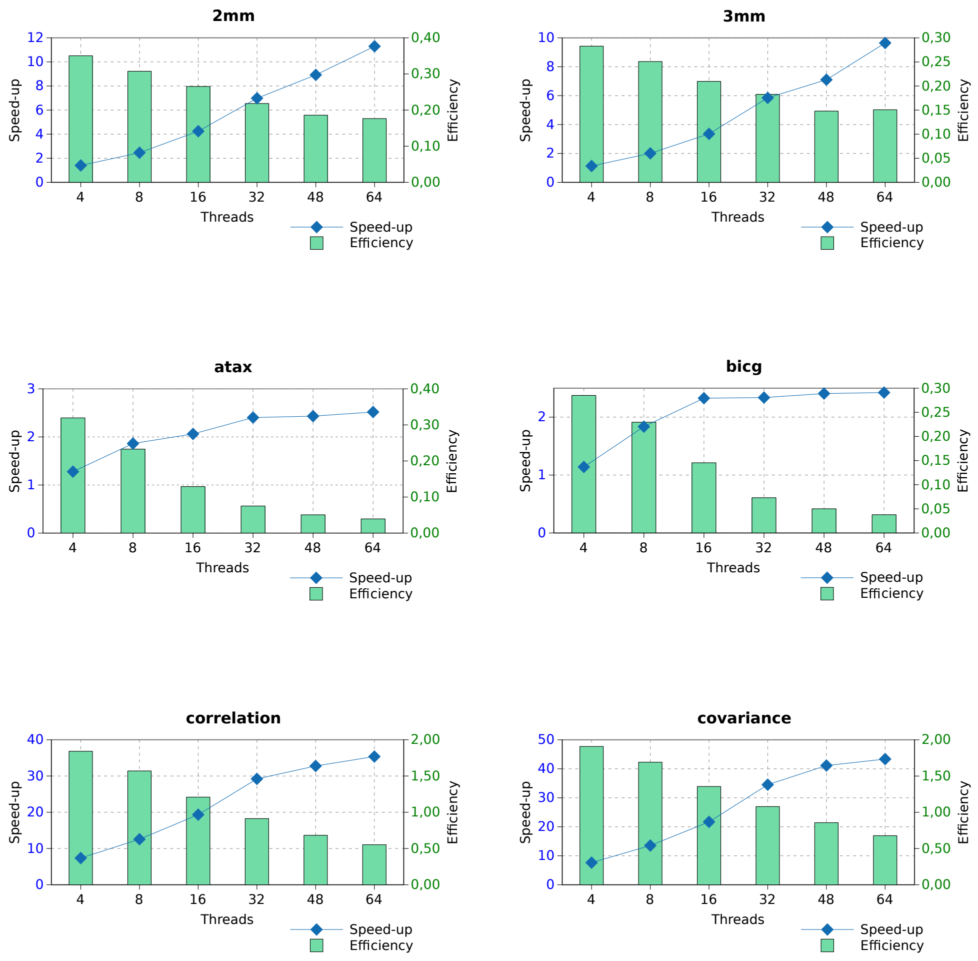

Figure 4. Speed-up and efficiency for parallel tiled code generated by TC.

The code generated by TC for the analyzed loop nests can be found at `http://tc-optimizer.sourceforge.net` in the *results* directory.

Based on the results obtained, we may conclude that dynamic free scheduling applied to the PolyBench kernels whose inter-tile dependence graphs are cycle-free, allows us to achieve significant parallel tiled code speed-up. For all examined kernels, even a small number of threads allows us to reduce the overall execution time. The experiments carried out prove that the time overhead of run-time computations required to implement dynamic free scheduling is low and does not prevent us from achieving considerable speed-up of parallel tiled code for which a corresponding inter-tile graph is cycle-free.

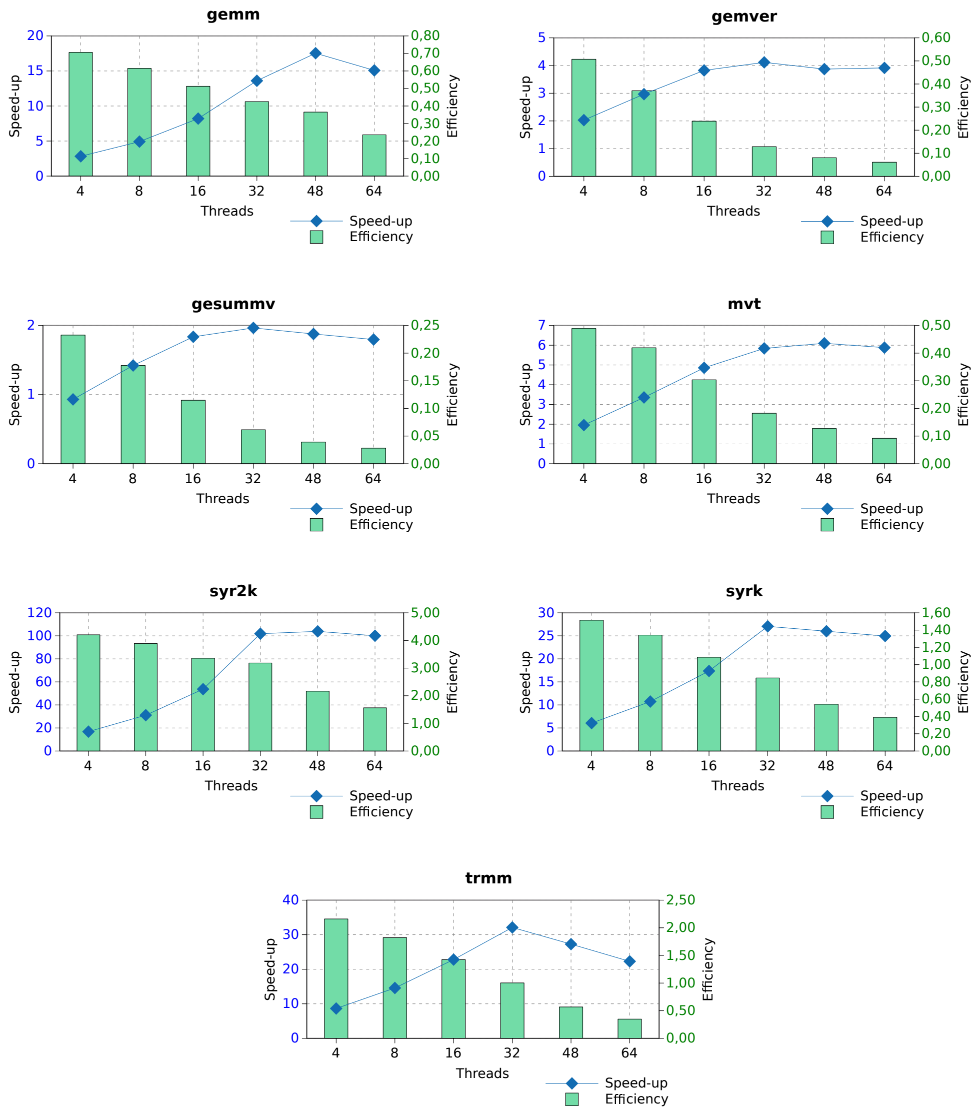

Figure 5. Speed-up and efficiency for parallel tiled code generated by TC.

5. Related work

The presented approach is within the data flow computation framework. In data flow, a computation can be represented by a directed graph. The nodes of the graph are operators and the arcs represent data paths. An arc into a node is an input operand path; an arc leaving a node is a result path. Execution of a data flow graph is based on operand availability at each node. Systolic arrays [8] and Maxeler dataflow machines [15] exploit this framework. The main differences between the presented approach and Systolic arrays and Maxeler dataflow machines are as follows. In the presented approach, a data flow graph is formed at run-time, while in Systolic arrays and Maxeler machines, it is formed at compile-time and then it is mapped into low-level hardware. The presented approach is to be implemented in general-purpose computer systems, while Systolic arrays and Maxeler machines are problem-specific ones. To our best knowledge, the MaxCompiler of Maxeler machines does not provide any loop nest tiling transformation.

Dynamic scheduling has already been recognized as a powerful and scalable alternative to well-known tiling techniques aimed at generation of static code. Most of common approaches are based on the *Affine Transformation Framework*. Paper [1] proposes a priority queue-based task scheduler, with a scheduling strategy governed by a critical path analysis. The priority metric of each task is computed as a length of the longest path in an associated inter-tile dependence graph, which starts with the prioritized vertex and ends with a leaf (a node with no successors). Tasks are enqueued as soon as their dependences are resolved. Similarly to that approach, our algorithm requires an inter-tile acyclic dependence graph for finding a legal order of tiles execution. By contrast, we do not consider any affine transformations for tiled code generation, an inter-tile dependence graph is formed statically, and we exploit the free schedule calculated at run-time.

Paper [9] presents *DynTile*, the system that employs dynamic scheduling for parallel execution of parametric tiles; i.e., it uses a dynamic scheduler to extract dependences among tiles whose sizes are run-time parameters. The algorithm discussed in that paper pre-processes the input loop nest to enable tiling and then applies wavefront processing of the computed rectangular tiles. In contrast to that technique, our approach does not require finding any affine transformation to enable tiling; instead of wavefronting based on affine transformations, we find and apply the free schedule at run-time.

Papers [16, 17] introduce tiling methodology for sparse matrix computations. Effective processing of sparse matrices requires the introduction of a memory-efficient data structure, *compressed sparse row* (CSR), which includes only nonzero values from the corresponding matrix. This entails non-affine loop bounds and indirect memory references (an index is expressed as a value of another memory location, unknown until run-time), which inhibits the application of any compile-time transformation aimed at data locality enhancement. The algorithm is based on the inspector/executor framework for performing run-time iteration reordering, i.e., "For sparse tiling, the inspector examines the non-zero structure of the sparse matrix at run-time, generates a data reordering and a schedule based on a tiling function, and remaps the sparse matrix and vectors based on the data reordering. The executor is a transformed version of the original code that uses the remapped matrix and vectors and the schedule created by the inspector" [17].

Free scheduling for arbitrarily nested loops was studied in paper [3]. That technique extracts statically fine-grained parallelism based on calculating the power k of a relation representing dependences in a loop nest. In contrary to that approach, the algorithm introduced in the present paper extracts coarse-grained parallelism and does not require any representation of the power k of a dependence relation.

Mullapudi and Bondhugula [12] suggested checking 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. But the authors do not suggest any way allowing for automatic generation of code implementing dynamic scheduling, the two programs presented were tiled manually.

The technique of forming and representing tiles by means of a parameterized set, that the presented approach builds on, was proposed in paper [2]. However, that paper focuses on statically generated tiled code. The technique was extended in paper [14] which describes an approach for extracting synchronization-free slices composed of tiles.

6. Conclusions

In this paper, we have presented a tiling approach for locality enhancement and extraction of coarse-grained parallelism from arbitrarily nested parameterized affine loops. The main merit of the introduced approach is that it does not require neither any affine transformation nor dependence graph transitive closure to find the free schedule at run-time when an inter-tile dependence graph based on original rectangular tiles is cycle-free.

This considerably reduces the computational complexity of the introduced dynamic tile schedule when compared to that of well-known techniques. As a consequence, the approach allows for achieving considerable speed-up of parallel codes for tiled loop nests with cycle-free inter-tile dependence graphs. Techniques aimed at breaking cycles in the original inter-tile dependence graph and then calculating the free schedule at run-time will be addressed in our future publications.

References

[1] Baskaran M.M., Vydyanathan N., Bondhugula U.K.R., Ramanujam J., Rountev A., Sadayappan P.: Compiler-Assisted Dynamic Scheduling for Effective Parallelization of Loop Nests on Multicore Processors. In: *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pp. 219–228, ACM, 2009.

[2] Bielecki W., Pałkowski M.: Perfectly Nested Loop Tiling Transformations Based on the Transitive Closure of the Program Dependence Graph. In: *Soft Computing in Computer and Information Science*, *Advances in Intelligent Systems and Computing*, vol. 342, pp. 309–320, Springer International Publishing, 2015.

[3] Bielecki W., Pałkowski M., Klimek T.: Free Scheduling for Statement Instances of Parameterized Arbitrarily Nested Affine Loops, *Parallel Computing*, vol. 38(9), pp. 518–532, 2012. `http://dx.doi.org/10.1016/j.parco.2012.06.001`.

[4] Darte A., Robert Y., Vivien F.: *Scheduling and Automatic Parallelization*, Lecture Notes in Computer Science, Birkhäuser, Boston, 2000.

[5] Feautrier P.: Some Efficient Solutions to the Affine Scheduling Problem. Part I. One-dimensional Time. In: *International Journal of Parallel Programming*, vol. 21(5), pp. 313–348, 1992. `http://dx.doi.org/10.1007/BF01407835`.

[6] Feautrier P.: Some Efficient Solutions to the Affine Scheduling Problem. Part II. Multidimensional Time. In: *International Journal of Parallel Programming*, vol. 21(6), pp. 389–420, 1992. `http://dx.doi.org/10.1007/BF01379404`.

[7] Feautrier P., Lengauer C.: *Encyclopedia of Parallel Computing*, chap. Polyhedron Model, pp. 1581–1592, Springer US, 2011. `http://dx.doi.org/10.1007/978-0-387-09766-4_502`.

[8] Gusev M., Evans D.J.: A new matrix vector product systolic array, *Journal of Parallel and Distributed Computing*, vol. 22(2), pp. 346–349, 1994.

[9] Hartono A., Baskaran M.M., Ramanujam J., Sadayappan P.: DynTile: Parametric Tiled Loop Generation for Parallel Execution on Multicore Processors. In: *Proceedings of the 24th IEEE International Symposium on Parallel and Distributed Processing*, pp. 1–12, 2010. `http://dx.doi.org/10.1109/IPDPS.2010.5470459`.

[10] Irigoin F., Triolet R.: Supernode Partitioning. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pp. 319–329, ACM, 1988. `http://dx.doi.org/10.1145/73560.73588`.

[11] Kelly W., Pugh W., Rosser E., Shpeisman T.: Transitive closure of infinite graphs and its applications, *International Journal of Parallel Programming*, vol. 24(6), pp. 579–598, 1996.

[12] Mullapudi R.T., Bondhugula U.: Tiling for Dynamic Scheduling. In: *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques*, Vienna, Austria, 2014.

[13] OpenMP Application Program Interface, Version 3.0, 2008. `http://www.openmp.org/mp-documents/spec30.pdf` [accessed 01 February 2016].

[14] Pałkowski M., Klimek T., Bielecki W.: TRACO: An Automatic Loop Nest Parallelizer for Numerical Applications. In: *Computer Science and Information Systems (FedCSIS), 2015 Federated Conference on Computer Science and Information Systems*, pp. 681–686, 2015.

[15] Pell O., Averbukh V.: Maximum performance computing with dataflow engines, *Computing in Science & Engineering*, vol. 14(4), pp. 98–103, 2012.

[16] Strout M.M., Carter L., Ferrante J.: Rescheduling for Locality in Sparse Matrix Computations. In: *Computational Science – ICCS 2001*, *Lecture Notes in Computer Science*, vol. 2073, pp. 137–146. Springer, Berlin–Heidelberg, 2001. `http://dx.doi.org/10.1007/3-540-45545-0\_23`.

[17] Strout M.M., Carter L., Ferrante J., Kreaseck B.: Sparse Tiling for Stationary Iterative Methods, *International Journal of High Performance Computing Applications*, vol. 18(1), pp. 95–113, 2004. `http://dx.doi.org/10.1177/1094342004041294`.

[18] The Polyhedral Benchmark suite, 2015. `http://web.cse.ohio-state.edu/~pouchet/software/polybench` [accessed 01 February 2016].

[19] Verdoolaege S.: isl: An Integer Set Library for the Polyhedral Model. In: *Mathematical Software – ICMS 2010*, *Lecture Notes in Computer Science*, vol. 6327, pp. 299–302. Springer, Berlin–Heidelberg, 2010. `http://dx.doi.org/10.1007/978-3-642-15582-6\_49`.

[20] Verdoolaege S.: *Integer Set Library: Manual, Version isl-0.16*, 2016. `http://isl.gforge.inria.fr/manual.pdf` [accessed 01 February 2016].

[21] Verdoolaege S.: *Presburger Formulas and Polyhedral Compilation, v0.02*. Polly Labs and KU Leuven, 2016.

[22] Verdoolaege S., Grosser T.: Polyhedral Extraction Tool. In: *Proceedings of the 2nd International Workshop on Polyhedral Compilation Techniques*. Paris, France, 2012.

Affiliations

Włodzimierz Bielecki
West Pomeranian University of Technology, Faculty of Computer Science, Szczecin, Poland, `wbielecki@wi.zut.edu.pl`

Piotr Skotnicki
West Pomeranian University of Technology, Faculty of Computer Science, Szczecin, Poland, `pskotnicki@wi.zut.edu.pl`

Received: 18.07.2016
Revised: 16.12.2016
Accepted: 18.12.2016