Free Scheduling of Tiles Based on the Transitive Closure of Dependence Graphs

Wlodzimierz Bielecki(✉), Marek Palkowski, and Tomasz Klimek

Faculty of Computer Science and Information Systems,
West Pomeranian University of Technology in Szczecin,
Zolnierska 49, 71210 Szczecin, Poland
{wbielecki,mpalkowski}@wi.zut.edu.pl
http://www.wi.zut.edu.pl

Abstract. A novel approach to form the free schedule of tiles comprising statement instances of the program loop nest is presented. Forming both valid tiles and free scheduling are based on the transitive closure of loop nest dependence graphs. Under the free schedule, tiles are executed as soon as their operands are available. To describe and implement the approach, loop dependences are presented in the form of tuple relations. A discussed algorithm is implemented in the open source TRACO compiler. Experimental results exposing the effectiveness of the introduced algorithm and speed-up of parallel programs, produced by means of this algorithm, are discussed.

Keywords: Loop nest tiling · Transitive closure · Dependence graphs · Coarse-grained parallelism · Free scheduling

1 Introduction

Tiling is a very important iteration reordering transformation for both improving data locality and extracting loop parallelism. Loop tiling for improving locality groups loop statement instances in a loop iteration space into smaller blocks (tiles) allowing reuse when the block fits in local memory. On the basis of a valid schedule of tiles, parallel coarse-grained code can be generated.

To our best knowledge, well-known tiling techniques are based on linear or affine transformations of program loop nests [6,9,10,13,20]. In paper [5], we describe the limitations of affine transformations and present how the free-scheduling of loop nest statement instances can be formed by means of the transitive closure of program dependence graphs. In this paper, we demonstrate how the approach, presented in our paper [5], can be adapted to form the free-scheduling of valid tiles. To generate both valid tiles and free-scheduling, we apply the transitive closure of dependence graphs. The proposed approach allows generation of parallel tiled code even when there does not exist an affine transformation allowing for producing a fully permutable loop nest. This approach is a result of a combination of the polyhedral model and the iteration space slicing framework.

R. Wyrzykowski et al. (Eds.): PPAM 2015, Part II, LNCS 9574, pp. 133–142, 2016.
DOI: 10.1007/978-3-319-32152-3_13

2 Background

A considered approach uses the dependence analysis proposed by Pugh and Wonnacott [16] where dependences are represented by dependence relations. Dependences of a loop nest are described by dependence relations with constraints presented by means of the Presburger arithmetic.

A dependence relation is a tuple relation of the form [*input list*]→[*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 list* and *output list* and it is a Presburger formula built of constraints represented with algebraic expressions and using logical and existential operators. A dependence relation is a mathematical representation of a data dependence graph whose vertices correspond to loop statement instances while edges connect dependent instances. The input and output tuples of a relation represent dependence sources and destinations, respectively; the relation constraints point out instances which are dependent.

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. $e \rightarrow e' \in R, e \in S$). In detail, the description of these operations is presented in papers [11,16].

The positive transitive closure for a given relation R, R^+, is defined as follows [11]: $R^+ = \{e \rightarrow e' : \ e \rightarrow e' \in R \vee \exists e'' s.t.\ e \rightarrow e'' \in R \wedge e'' \rightarrow e' \in R^+\}$. 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 [12]: $R^* = R^+ \cup I$, where I the identity relation. It describes the same connections in a dependence graph (represented by R) that R^+ does plus connections of each vertex with itself.

The composition of given relations $R_1 = \{x_1 \rightarrow y_1 | f_1(x_1, y_1)\}$ and $R_2 = \{x_2 \rightarrow y_2 | f_2(x_2, y_2)\}$, is defined as follows [11]: $R_1 \circ R_2 = \{x \rightarrow y | \exists z \ s.t.\ f_1(z, y) \wedge f_2(x, z)\}$.

3 Finding Free Scheduling

The algorithm, presented in our paper [5], allows us to generate fine-grained parallel code based on the free schedule representing time partitions; all statement instances of a time partition can be executed in parallel, while partitions are enumerated sequentially. The free schedule function is defined as follows.

Definition 1 [7,8]. The *free schedule* is the function that assigns discrete time of execution to each loop nest statement instance as soon as its operands are available, that is, it is mapping $\sigma : LD \rightarrow \mathbb{Z}$ such that

$$\sigma(p) = \begin{cases} 0 \ if \ there \ is \ no \ p_1 \in LD \ s.t. \ p_1 \rightarrow p \\ 1 + max(\sigma(p_1), \sigma(p_2), ..., \sigma(p_n)); p, p_1, p_2, ..., p_n \in LD; \\ p_1 \rightarrow p, p_2 \rightarrow p, ..., p_n \rightarrow p, \end{cases}$$

where $p, p_1, p_2, ..., p_n$ are loop nest statement instances, LD is the loop nest domain, $p_1 \rightarrow p, p_2 \rightarrow p, ..., p_n \rightarrow p$ mean that the pairs p_1 and p, p_2 and p, ...,p_n and p are dependent, p represents the destination while $p_1, p_2, ..., p_n$ represent the sources of dependences, n is the number of operands of statement instance p (the number of dependences whose destination is statement instance p).

The free schedule is the fastest legal schedule [8]. In paper [5] we presented fine-grained parallelism extraction based on the power k of relation R.

The idea of the algorithm is the following [5]. Given relations $R_1, R_2, ..., R_m$, representing all dependences in a loop nest, we first calculate $R = \bigcup_{i=1}^{m} R_i$ and then R^k, where $R^k = \underbrace{R \circ R \circ ...R}_{k}$, "$\circ$" is the composition operation. Techniques of calculating the power k of relation R are presented in the following publications [12,17] and they are out of the scope of this paper. Let us only note that given transitive closure R^+, we can easily convert it to the power k of R, R^k, and vice versa, for details see [17].

Given set UDS comprising all loop nest statement instances that are ready to execution at time $k = 0$ (Ultimate Dependence Sources), each vertex, represented with the set $S_k = R^k(UDS) - R^+ \circ R^k(UDS)$, is connected in the dependence graph, defined by relation R, with some vertex(ices) represented by set UDS with a path of length k. Hence at time k, all the statement instances belonging to the set S_k can be scheduled for execution and it is guaranteed that k is as few as possible.

4 Loop Nest Tiling Based on the Transitive Closure of Dependence Graphs

In this paper, to generate valid tiled code, we apply the approach presented in paper [4], which is based on the transitive closure of dependence graphs. Next, we briefly present the steps of that approach.

First, we form set $TILE(\boldsymbol{II}, \boldsymbol{B})$, including iterations belonging to a parametric tile, as follows $TILE(\boldsymbol{II}, \boldsymbol{B}) = \{[\boldsymbol{I}] | \boldsymbol{B}^*\boldsymbol{II} + \boldsymbol{LB} \leq \boldsymbol{I} \leq \min(\boldsymbol{B}^*(\boldsymbol{II} + \mathbf{1}) + \boldsymbol{LB} - \mathbf{1}, \boldsymbol{UB})$ AND $\boldsymbol{II} \geq 0\}$, where vectors $\boldsymbol{LB}$ and $\boldsymbol{UB}$ include the lower and upper loop index bounds of the original loop nest, respectively; diagonal matrix $\boldsymbol{B}$ defines the size of a rectangular original tile; elements of vector $\boldsymbol{I}$ represent the original loop nest iterations contained in the tile whose identifier is $\boldsymbol{II}$; $\mathbf{1}$ is the vector whose all elements have value 1; here and further on, the notation $x \geq (\leq) y$ where x, y are two vectors in $\mathbb{Z}^n$ corresponds to the component-wise inequality, that is, $x \geq (\leq) y \Longleftrightarrow x_i \geq (\leq) y_i$, i=1,2,...,n.

Next, we build sets $TILE\_LT$ and $TILE\_GT$ that are the unions of all the tiles whose identifiers are lexicographically less and greater than that of $TILE(\boldsymbol{II}, \boldsymbol{B})$, respectively:

$TILE\_LT = \{[\boldsymbol{I}] \mid$ exists $\boldsymbol{II'}$ s. t. $\boldsymbol{II'} \prec \boldsymbol{II}$ AND $\boldsymbol{II} \geq 0$ AND $\boldsymbol{B}^*\boldsymbol{II} + \boldsymbol{LB} \leq \boldsymbol{UB}$ AND $\boldsymbol{II'} \geq 0$ and $\boldsymbol{B}^*\boldsymbol{II'} + \boldsymbol{LB} \leq \boldsymbol{UB}$ AND $\boldsymbol{I}$ in $TILE(\boldsymbol{II'}, \boldsymbol{B})\}$,

$TILE\_GT = \{[\boldsymbol{I}] \mid \text{exists } \boldsymbol{II}' \text{ s. t. } \boldsymbol{II}' \succ \boldsymbol{II} \text{ AND } \boldsymbol{II} \geq 0 \text{ AND } \boldsymbol{B}^*\boldsymbol{II}+\boldsymbol{LB} \leq \boldsymbol{UB} \text{ AND } \boldsymbol{II}' \geq 0 \text{ and } \boldsymbol{B}^*\boldsymbol{II}' + \boldsymbol{LB} \leq \boldsymbol{UB} \text{ AND } \boldsymbol{I} \text{ in } TILE(\boldsymbol{II}', \boldsymbol{B})\}$,
where "$\prec$" and "$\succ$" (here and further on) denote the lexicographical relation operators for two vectors. Then, we calculate set

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

which does not include any invalid dependence target, i.e., it does not include any dependence target whose source is within set $TILE\_GT$. The following set

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

includes all the iterations that (i) belong to the tiles whose identifiers are lexicographically less than that of set $TILE\_ITR$, (ii) are the targets of the dependences whose sources are contained in set $TILE\_ITR$, and (iii) are not any target of a dependence whose source belong to set $TILE\_GT$. Target tiles are defined by the following set $TILE\_VLD = TILE\_ITR \cup TVLD\_LT$.

Lastly, we form set $TILE\_VLD\_EXT$ by means of inserting (i) into the first positions of the tuple of set $TILE\_VLD$ elements of vector $\boldsymbol{II}$: $ii_1, ii_2, ..., ii_d$; (ii) into the constraints of set $TILE\_VLD$ the constraints defining tile identifiers $\boldsymbol{II} \geq 0$ and $\boldsymbol{B}^*\boldsymbol{II} + \boldsymbol{LB} \leq \boldsymbol{UB}$. Target code is generated by means of applying any code generator allowing for scanning elements of set $TILE\_VLD\_EXT$ in the lexicographic order, for example, CLooG [1].

5 Free Scheduling for Tiles

The algorithm presented in this paper is a combination of the approaches presented in the two previous sections. First, we generate tiled code as it is described in Sect. 4, then we find free scheduling for tiles of the tiled code. For this purpose, first, we form relation, R_TILE, which describes dependences among tiles as follows

R_TILE:=$\{[\boldsymbol{II}]->[\boldsymbol{JJ}]$: exist $\boldsymbol{I}, \boldsymbol{J}$ s.t. $(\boldsymbol{II},\boldsymbol{I})$ in $TILE\_VLD\_EXT(\boldsymbol{II})$ AND $(\boldsymbol{JJ},\boldsymbol{J})$ in $TILE\_VLD\_EXT_i(\boldsymbol{JJ})$ AND $\boldsymbol{J}$ in $R(\boldsymbol{I})\}$,
where $\boldsymbol{II}$, $\boldsymbol{JJ}$ are the vectors representing tile identifiers; vectors $\boldsymbol{I}$, $\boldsymbol{J}$ comprise iterations belonging to tiles whose identifiers are $\boldsymbol{II}$, $\boldsymbol{JJ}$, respectively.

The following step is to calculate set, UDS, including the tile identifiers which state for tile ultimate dependence sources and/or independent ones as follows: UDS=II_SET − range (R_TILE), where set $II\_SET = \{[\boldsymbol{II}] | \boldsymbol{II} \geq 0 \text{ and } \boldsymbol{B}^*\boldsymbol{II} + \boldsymbol{LB} \leq \boldsymbol{UB}\}$ represents all tile identifiers.

Now, we apply the algorithm presented in paper [5] to form free-scheduling for tiles of tiled code. With this purpose, we calculate the transitive closure and power k of relation R_TILE and next calculate set S_k, representing the free schedule, as follows $S_k = R\_TILE^k(UDS) - (R\_TILE^+ \circ R\_TILE^k(UDS))$. Finally, we extend the tuple of set S_k with variable k and variables representing statement instances of a parametric target tile(together with corresponding constraints) and generate code applying any code generator, for example, CLooG to scan iterations within set S_k in the lexicographical order. Algorithm 1 presents the discussed above idea in a formal way.

Algorithm 1. Parallel tiled code generation based on the free schedule

Input: A loop nest of depth d; constants $b_1, b_2, ..., b_d$ defining the size of a rectangular original tile, relation R representing all the dependences in the loop nest.
Output: Code enumerating time partitions according to the free schedule, tiles for each time partition (in parallel), and statement instances in each tile.
Method:

1. Apply the algorithm, presented in paper [4] to the original loop nest to generate sets *II_SET*, *TILE_VLD*, *TILE_VLD_EXT*, and tiled code.
2. Form relation, R_TILE, which describes dependences among tiles but ignores dependences within each tile as follows

   R_TILE:={[***II***]->[***JJ***]: exist ***I***, ***J*** s.t. (***II,I***) in *TILE_VLD_EXT*(***II***) *AND* (***JJ,J***) in *TILE_VLD_EXT*(***JJ***) *AND* ***J*** in R(***I***)},
   where ***II, JJ*** are the vectors representing tile identifiers, $TILE\_VLD\_EXT$ is the set returned by step 1.
3. Calculate set, *UDS*, including the tile identifiers which state for tile ultimate dependence sources and/or independent ones as follows
   UDS:=*II_SET* − range (R_TILE),
4. Calculate set
   S_k= $R\_TILE^k(UDS) - (R\_TILE^+ \circ R\_TILE^k(UDS))$.
5. Extend set S_k as follows: insert in the first its tuple position symbolic variable k responsible for representing time under the free schedule; insert in the last its tuple positions the elements of set $TILE\_VLD$ returned by step 1; insert into the constraint of set S_k the constraint of set $TILE\_VLD$.
6. Apply to the set, returned by step 5, CLooG [1] and postprocess the code generated by CLooG to get the following code structure

```
seqfor for k  //enumerating time partitions
  parfor Sk //enumerating tile identifiers contained in set Sk
            //formed in step 4  for a given value of k
     seqfor TILE_VLD //enumerating  statement instances comprised in
               //set TILE_VLD defined by the tile identifiers
               //represented  by the previous parfor loop
```

6 Illustrative Example

In this section, we illustrate steps of Algorithm 1 by means of the following loop:

```
for(i=1; i<=6; i++)
 for(j=1; j<=6; j++)
  a[i][j] = a[i+1][j-1];
```

We use the ISL library to carry out operations on relations and sets required by the presented algorithm. A dependence relation, returned by Petit, the Omega project dependence analyzer, is the following

```
R:= {[i,j,v] -> [i',j',v'] : ( i' = 1+i and j' = j-1 and v = 6 and v' = 6
and 1 <= i <= 5 and 2 <= j <= 6 )},
```

where here and further on "6" states for the statement identifier represented via the corresponding line number in the original loop nest.

The algorithm presented in paper [4] returns the following set $TILE\_VLD\_EXT$ representing both tile identifiers and statement instances within each target tile.

```
TILE_VLD_EXT:= { [i0, i1, i2, i3, 6] : i0 >= 0 and i2 >= 1 + 2i0 and
i2 <= 6 and i3 >= 1 + 2i1 and i3 <= 6 and i3 >= 1 and i3 <= 3 + 2i0 +
2i1 - i2; [i0, i1, 2 + 2i0, 2i1, 6] : i0 <= 2 and i0 >= 0 and i1 <= 2
and i1 >= 1; [i0, 2, 2 + 2i0, 6, 6] : i0 <= 2 and i0 >= 0 }.
```

Using relation R and set $TILE\_VLD\_EXT$, we form realtion R_TILE that is of the form below.

```
R_TILE:= { [i0, i1, 6] -> [1 + i0, -1 + i1, 6] : i0 >= 0 and i0 <= 1 and
i1 <= 2 and i1 >= 1; [i0, 2, 6] -> [1 + i0, 2, 6] : i0 <= 1 and i0 >= 0 }.
```

Set UDS is the following $\{[0,\ jj, 6] : jj \leq 2 \text{ and } jj \geq 0\}$.

Using the appropriate functions of the ISL library to calculate relations R_TILE^k and R_TILE^+, we calculate set S_k according to the formula in step 4 of Algorithm 1, and extend set S_k as presented in step 5 of Algorithm 1, to get:

```
Sk:= { [i0, i0, i2, i3, i4, 6] : i3 >= 1 + 2i0 and i4 >= 1 + i2 and
i2 <= 2 and i3 <= 2 + 2i0 and i0 >= 0 and i4 <= 2 + 2i2 and i4 >= 2 +
2i0 + 2i2 - i3 and i0 <= 2 and 2i4 <= 6 + 4i0 + 5i2 - 2i3 }.
```

Finally, we apply to set S_k the GLooG code generator and postprocess the code returned by CLooG to yield the following OpenMP C code.

```
1. for (c0 = 0; c0 <= 2; c0 += 1)
2. #pragma omp parallel for
3.   for (c2 = 0; c2 <= 2; c2 += 1)
4.     for (c3 = 2 * c0 + 1; c3 <= 2 * c0 + 2; c3 += 1)
5.       for (c4 = max(c2 + 1, 2 * c0 + 2 * c2 - c3 + 2);
          c4 <= min(2*c0 + 2 * c2 - c3 + c2/2 + 3, 2 * c2 + 2); c4++)
6.           a[c3][c4]=a[c3+1][c4-1];
```

where line 1 presents the serial *for* loop enumerating time partitions; line 2 represents the two OpenMP directives (*parallel for*) pointing out that the iterations of the *for* loop in line 3 can be executed in parallel; the *for* loops in line 1 and line 3 enumerate tile identifiers, whereas the *for* loops in line 4 and line 5 scan iterations within a tile. Figure 1 presents original tiles, while Fig. 2 shows target tiles returned by the algorithm, presented in paper [4] (depicted by dashed lines), and the three time partitions (k=0, 1, 2) for the illustrative example.

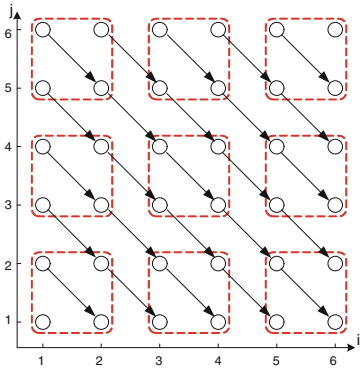

Fig. 1. Original tiles

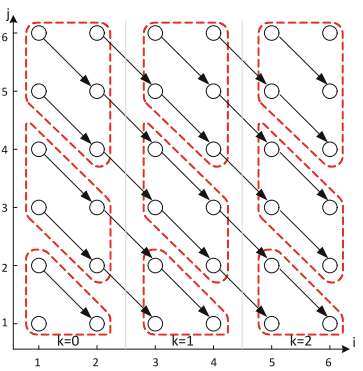

Fig. 2. Target tiles and time partitions

7 Experimental Study

The presented algorithm has been implemented in the optimizing compiler TRACO, publicly available at the website http://traco.sourceforge.net. For calculating R^+ and R^k, TRACO uses the corresponding functions of the ISL library [17]. To evaluate the effectiveness of proposed approach, we have experimented with NAS Parallel Benchmarks 3.3 (NPB) [14].

From 431 loops of the NAS benchmark suite, Petit is able to analyse 257 loops, and dependences are available in 134 loops (the rest 123 loops do not expose any dependence). For these 134 loop nests, ISL is able to calculate R_TILE^k for 58 ones and accordingly TRACO is able to generate parallel tiled code for those programs. Such a limitation is not the limitation of the algorithm, it is the limitation of the corresponding ISL function.

To check the performance of parallel tiled code, produced with TRACO, the following criteria were taken into account for choosing NAS programs: (i) a loop nest must be computationally intensive (there are many NAS benchmarks with constant upper bounds of loop indices, hence their parallelization is not justified), (ii) structures of chosen loops must be different (there are many loops of a similar structure).

Applying these criteria, we have selected the following five NAS loops: *BT_rhs_1* (Block Tridiagonal Benchmark), *FT_auxfnct.f2p_2* (Fast Fourier Transform Benchmark), *UA_diffuse_5*, *UA_setup_16* and *UA_transfer_4* (Unstructured Adaptive Benchmark).

To carry out experiments, we have used a computer with Intel i5-4670 3.40 GHz processors (Haswell, 2013), 6 MB cache and 8 GB RAM. Source and target codes of the examined programs are available in http://sourceforge.net/p/issf/code-0/HEAD/tree/trunk/examples/fstile/.

Table 1 presents execution time and speed-up for the studied loop nests. Speed-up is the ratio of sequential and parallel program execution times, i.e., $S = T(1)/T(P)$, where $T(P)$ is the parallel program execution time on P processors. Speedups were computed against the serial original code execution time.

Table 1. Speed-up of parallel tiled loop nests for 4 CPU cores.

Program	Loop up. bounds	Time of serial run (in seconds)	Block size	Time of parallel run (in seconds)	Speed-up
FT_auxfnct.f2p_2	N1, N2, N3 = 500	6.857	16	0.817	8.393
			32	0.795	8.625
	N1, N2, N3 = 600	13.403	16	1.176	11.397
			32	1.228	10.914
BT_rhs.f2p_1	N1, N2, N3 = 200	2.87	16	0.892	3.217
			32	1.112	2.581
	N1, N2, N3 = 300	10.598	16	2.936	3.610
			32	3.549	2.986
UA_diffuse.f2p_5	N1, N2, N3, N4 = 100	0.444	16	0.209	2.124
			32	0.187	2.374
	N1, N2, N3, N4 = 200	10.875	16	3.85	2.825
			32	3.556	3.058
UA_setup.f2p_16	N1, N2, N3 = 1000	1.325	16	0.662	2.002
			32	0.445	2.978
	N1, N2, N3 = 1100	15.285	16	0.976	15.661
			32	0.746	20.489
UA_transfer.f2p_4	N1, N2, N3 = 700	5.541	16	0.742	7.468
			32	0.745	7.438
	N1, N2, N3 = 1000	22.751	16	1.501	15.157
			32	1.499	15.177

Experiments were carried out for 4 CPUs. Analysing the data in Table 1, we may conclude that for all parallel tiled loops, positive speed-up is achieved. It depends on the problem size defined by loop index upper bounds and a tile size. It is worth to note that for the *FT_auxfnct.f2p_2* and *UA_transfer_4* programs, super-linear speed-up is achieved, i.e., the speed-up is greater than 4 – the number of CPUs used. This phenomenon could be explained by the fact that the data size required by the original program is greater than the cache size when executed sequentially, but could fit nicely in each available cache when executed in parallel, i.e., due to increasing program locality.

8 Related Work

There has been a considerable amount of research into tiling demonstrating how to aggregate a set of loop iterations into tiles with each tile as an atomic macro statement, starting with pioneer paper [10] and those presenting advanced techniques [6,9,19].

One of the most advanced reordering transformation frameworks is based on the polyhedral model. Let us remind that "*Restructuring programs using the polyhedral model is a three steps framework. First, the Program Analysis phase aims at translating high level codes to their polyhedral representation and to provide data dependence analysis based on this representation. Second, some*

optimizing or parallelizing algorithm uses the analysis to restructure the programs in the polyhedral model. This is the Program Transformation step. Lastly, the Code Generation step returns back from the polyhedral representation to a high level program" [3].

All above three steps are available in the approach presented in this paper. But there exists the following difference in step 2: in the polyhedral model "*a (sequence of) program transformation(s) is represented by a set of affine functions, one for each statement*" [3] while the presented approach does not find and use any affine function. It applies the transitive closure of a program dependence graph to specific subspaces of the source loop iteration space. At this point of view the program transformation step is rather within the Iteration Space Slicing Framework introduced by Pugh and Rosser [15], where the key step is calculating the transitive closure of a program dependence graph.

Papers [10,18] are a seminal work presenting the theory of tiling techniques based on affine transformations. These papers present techniques consisting of two steps: they first transform the original loop into a fully permutable loop nest, then transform the fully permutable loop nest into tiled code. Loop nests are fully permutable if they can be permuted arbitrarily without altering the semantics of the source program. If a loop nest is fully permutable, it is sufficient to apply a tiling transformation to this loop nest [18].

Papers [2,5] demonstrate how we can extract coarse- and fine-grained parallelism applying different Iteration Space Slicing algorithms, however they do not consider any tiling transformation.

Wonnacott and Strout review implemented and proposed techniques for tiling dense array codes in an attempt to determine whether or not the techniques permit on scalability. They write [19]: "*No implementation was ever released for iteration space slicing*". This permits us to state that TRACO, which implements the algorithm, presented in this paper, is the first compiler where Iteration Space Slicing is applied to produce parallel tiled code based on the free-schedule of tiles.

9 Conclusion

In this paper, we presented a novel approach based on a combination of the Polyhedral Model and the Iteration Space Slicing framework. It allows generation of parallel tiled codes which demonstrate significant speed-up on shared memory machines with multi-core processors. The usage of the free schedule of tiles instead of that of loop nest statement instances allows us to adjust the parallelism grain-size to match the inter-processor communication capabilities of the target architecture. In the future, we plan to present an extended approach allowing for tiling with parallelepiped original tiles.

References

1. Bastoul, C.: Code generation in the polyhedral model is easier than you think. In: PACT 2013 IEEE International Conference on Parallel Architecture and Compilation Techniques, Juan-les-Pins, pp. 7–16, September 2004

2. Beletska, A., Bielecki, W., Cohen, A., Palkowski, M., Siedlecki, K.: Coarse-grained loop parallelization: iteration space slicing vs affine transformations. Parallel Comput. **37**, 479–497 (2011)
3. Benabderrahmane, M.-W., Pouchet, L.-N., Cohen, A., Bastoul, C.: The polyhedral model is more widely applicable than you think. In: Gupta, R. (ed.) CC 2010. LNCS, vol. 6011, pp. 283–303. Springer, Heidelberg (2010). http://dx.doi.org/10.1007/978-3-642-11970-5_16
4. Bielecki, W., Palkowski, M.: Perfectly nested loop tiling transformations based on the transitive closure of the program dependence graph. Soft Comput. Comput. Inf. Sci. **342**, 309–320 (2015)
5. Bielecki, W., Palkowski, M., Klimek, T.: Free scheduling for statement instances of parameterized arbitrarily nested affine loops. Parallel Comput. **38**(9), 518–532 (2012)
6. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. SIGPLAN Not. **43**(6), 101–113 (2008)
7. Darte, A., Khachiyan, L., Robert, Y.: Linear scheduling is nearly optimal. Parallel Process. Lett. **1**(2), 73–81 (1991)
8. Darte, A., Robert, Y., Vivien, F.: Scheduling and Automatic Parallelization. Birkhauser, New York (2000)
9. Griebl, M.: Automatic parallelization of loop programs for distributed memory architectures (2004)
10. Irigoin, F., Triolet, R.: Supernode partitioning. In: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1988, pp. 319–329. ACM, New York (1988)
11. Kelly, W., Maslov, V., Pugh, W., Rosser, E., Shpeisman, T., Wonnacott, D.: The omega library interface guide. Technical report, College Park, MD, USA (1995)
12. Kelly, W., Pugh, W., Rosser, E., Shpeisman, T.: Transitive closure of infinite graphs and its applications. Int. J. Parallel Program. **24**(6), 579–598 (1996)
13. Lim, A., Cheong, G.I., Lam, M.S.: An affine partitioning algorithm to maximize parallelism and minimize communication. In: Proceedings of the 13th ACM SIGARCH International Conference on Supercomputing, pp. 228–237. ACM Press (1999)
14. NAS benchmarks suite (2013). http://www.nas.nasa.gov
15. Pugh, W., Rosser, E.: Iteration space slicing and its application to communication optimization. In: International Conference on Supercomputing, pp. 221–228 (1997)
16. Pugh, W., Wonnacott, D.: An exact method for analysis of value-based array data dependences. In: Banerjee, U., Gelernter, D., Nicolau, A., Padua, D. (eds.) Languages and Compilers for Parallel Computing. LNCS, vol. 768, pp. 546–566. Springer, Heidelberg (1993)
17. Verdoolaege, S.: Integer set library - manual. Technical report (2011). http://www.kotnet.org/~skimo//isl/manual.pdf
18. Wolf, M.E., Lam, M.S.: A loop transformation theory and an algorithm to maximize parallelism. IEEE Trans. Parallel Distrib. Syst. **2**(4), 452–471 (1991)
19. Wonnacott, D.G., Strout, M.M.: On the scalability of loop tiling techniques. In: Proceedings of the 3rd International Workshop on Polyhedral Compilation Techniques (IMPACT), January 2013
20. Xue, J.: On tiling as a loop transformation (1997)