Coarse-Grained Loop Parallelization: Iteration Space Slicing vs Affine Transformations

Anna Beletska, Wlodzirmierz Bielecki, Albert Cohen, Marek Palkowski, Krzysztof Siedlecki

▶ To cite this version:

Anna Beletska, Wlodzirmierz Bielecki, Albert Cohen, Marek Palkowski, Krzysztof Siedlecki. Coarse-Grained Loop Parallelization: Iteration Space Slicing vs Affine Transformations. The 11th International Symposium on Parallel and Distributed Computing, Jul 2009, Munich, Germany. hal-00645329

HAL Id: hal-00645329

https://inria.hal.science/hal-00645329v1

Submitted on 27 Nov 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Coarse-Grained Loop Parallelization: Iteration Space Slicing vs Affine Transformations

Anna Beletska, INRIA Saclay, France, anna.beletska@inria.fr
Wlodzimierz Bielcki, Tech. Univ. of Szczecin, Poland, wbielecki@wi.ps.pl
Albert Cohen, INRIA Saclay, France, albert.cohen@inria.fr
Marek Palkowski, Tech. Univ. of Szczecin, Poland, mpalkowski@wi.ps.pl
Krzysztof Siedlecki, Tech. Univ. of Szczecin, Poland, ksiedlecki@wi.ps.pl

***Abstract*—Automatic coarse-grained parallelization of program loops is of great importance for multi-core computing systems. This paper presents a comparison of Iteration Space Slicing and Affine Transformation Framework algorithms aimed at extracting coarse-grained parallelism available in arbitrarily nested parameterized affine loops. We demonstrate that Iteration Space Slicing permits for extracting more coarse-grained parallelism in comparison to the Affine Transformation Framework. Experimental results show that by means of Iteration Space Slicing algorithms, we are able to extract coarse-grained parallelism for most loops of the NAS and UTDSP benchmarks, and that there is a strong need in devising advanced algorithms for calculating the exact transitive closure of dependence relations in order to increase the applicability of that framework.**

I. Introduction

The increasing use of multi-core microprocessors necessitates an increasing need to expose coarse-grained parallelism available in sequential applications. The lack of automated tools permitting for exposing such parallelism decreases the productivity of parallel programmers and increases the time and cost of producing a parallel program.

Because most computations are contained in program loops, automatic extraction of coarse-grained parallelism from loops is extremely important for multi-core systems, allowing us to produce parallel code from existing sequential applications and to create multiple threads that can be easily scheduled by a load balancer achieving a high system performance.

Currently, there exist different approaches to extract coarse-grained parallelism. In this paper, we discuss Iteration Space Slicing algorithms extracting coarse-grained parallelism from program loops [4]–[8] and compare them with Affine Transformation Framework algorithms [10]–[13], [16], [17]. Both classes of algorithms aim at parallelization of arbitrarily nested parameterized both uniform and non-uniform loops.

II. Background

A *statement instance* $s(I)$ is a particular execution of a statement s of the loop body for some loop iteration I.

Two statement instances $s_1(I)$ and $s_2(J)$ are dependent if both access the same memory location and if at least one access is a write. Provided that $s_1(I)$ is executed before $s_2(J)$, $s_1(I)$ and $s_2(J)$ are called the *source* and *destination* of the dependence, respectively. The sequential execution ordering of statement instances, denoted as $s_1(I) \prec s_2(J)$, is induced by the lexicographic ordering of iteration vectors and the textual ordering of statements when the instances share the same iteration vector.

Discussed coarse-grained parallelization algorithms deal with *static-control loop nests*, where lower and upper bounds as well as conditionals and array subscripts are affine functions of symbolic parameters and surrounding loop indices. They assume an exact representation of loop-carried dependences and consequently an exact dependence analysis which detects a dependence if and only if it actually exists at runtime between two given instances.[1] Any technique extracting exact dependences can be applied. We chose dependence analysis proposed by Pugh and Wonnacott [19] and implemented in Petit [25], where dependences are represented by relations whose constraints are described in the Presburger arithmetic (built of affine equalities and inequalities, logical and existential operators); the Omega calculator is used for computations over such relations [25].

Following Omega's conventions, a dependence relation is a tuple relation of the form

$$\{[input_list] \to [output_list] : \ constraints\},$$

where *input_list* and *output_list* are the lists of variables and/or expressions used to describe input and output tuples and *constraints* is a Presburger formula describing the constraints imposed upon *input_list* and *output_list*.

Further we assume that the relation is a single-conjunct relation [15]. Otherwise, it should be split into a set of the finit number of single-conjunct relations.

We distinguish between a dependence graph representing all the dependences among statement instances and a reduced dependence graph being composed by vertices for each statement of the loop s_i, $1 \le i \le r$, and edges joining vertices according to dependence relations $R_{i,j}$, $i,j \in [1,r]$, being exposed by an exact dependence analysis, where r is the number of statements within the loop body.

III. Affine Transformation Framework

The Affine Transformation Framework (ATF) unifies a large class of transformations, including loop interchange, reversal,

[1] A non-exact yet conservative (instancewise) representation of dependences is also possible, at the expense of parallelism extraction, while this work focuses on extracting the maximal degree of synchronization-free parallelism.

skewing, fusion, fission, reindexing, scaling and statement reordering [3], [10]–[13], [16], [17]. Affine transformations permit for the extraction of coarse-grained parallelism represented with synchronization-free threads. The basic idea of the technique assumes that statement instances of a loop are divided into partitions, such that dependent statement instances are placed in the same partition. A partitioning is described by an affine mapping for each loop statement. An *m-dimensional affine partition mapping* for a statement s in a loop is an m-dimensional affine expression $\Phi_s = C_s I_s + c_s$, which maps an instance of statement s, indexed by its iteration vector I_s, to an m-dimensional vector.

Affine transformations for each statement are found as follows. Given an m-statement loop that originates dependences presented with a set of n dependence relations $D = \{s_i(I_k) \rightarrow s_j(J_k) : \text{ constraints}_k\}$, where $i, j \in [1,2,...,m],\ k = 1,2,..,n$, for each statement s we have to find an m_s-dimensional affine space partition mapping $\Phi_s = C_s I_s + c_s$ such that $\Phi_{s_i}(I_k) = \Phi_{s_j}(J_k) = P_m$, where s_i, s_j are the statements whose instances originate sources and destinations, respectively, represented with dependence relation $\{s_i(I_k) \rightarrow s_j(J_k) : \text{ constraints}_k\}$, C_s is a matrix of dimensions $m_s \times n$, c_s is an m_s-dimensional vector representing a constant term, and P_m is a vector representing the identifier of a processor to execute the dependebce sources and destinations identified by the dependence relation.

In order to satisfy the condition $\Phi_{s_i}(I_k) = \Phi_{s_j}(J_k)$, we use dependence relations $\{s_i(I_k) \rightarrow s_j(J_k) : \text{ constraints}_k\}$, $i, j \in [1,2,..,m]$, $k = 1,2,..,n$, to build a system of equations of the form

$$C_{s_i} I_k + c_{s_i} - C_{s_j} J_k - c_{s_j} = 0,\ k = 1,2,...n \qquad (1)$$

Next, we should find solutions $[C_{s_i}\ c_{s_i}]$ and $[C_{s_j}\ c_{s_j}]$ to system (1) valid for any I_k, J_k satisfying constraints$_k$, k=1,2,...,n.

Using ATF, coarse-grained parallelism may be found as follows [16], [17]. First, using dependence relations extracted for a loop, we form a reduced dependence graph and find all strongly connected components (SCC) in it. For each SCC, we form a set of dependence relations defining vertices and edges. Then, we build a constraint in the form of (1) for each SCC, using a corresponding set of dependence relations. Let the resulting system of linear equations be in the form AX=0, where X is a vector representing all the unknown coordinates of C_s and the constant terms c_s of affine mappings. The next step is to eliminate all constant terms c_s from AX=0, by using, for example, traditional Gaussian Elimination, to obtain a reduced system of equations of the form $A'X'$=0, where X' represents unknown C_s. We have to find a solution to $A'X'$=0 as a set of basic vectors spanning the null space of A'. From each of those vectors we find one row of an affine mapping - coefficients C_s are formed directly by a basic vector and constant terms c_s are found using AX=0.

Having found an affine mapping $\Phi_s = C_s I_s + c_s$ for each statement s, we can apply well-known techniques [1], [2], [9], [20], [21], for generating parallel code for each SCC. Resulting code is formed taking in the account the topology of SCCs in the reduced dependence graph.

IV. Iteration Space Slicing

Iteration Space Slicing (ISS) was introduced by Pugh and Rosser [18] as an extension of a program slicing proposed by Weiser [22]. It takes dependence information as input to find all statement instances that must be executed to produce the correct values for the specified array elements. A dependence graph refers to the extensive set of dependences of a loop nest, described by dependence relations in the Presburger arithmetic. The algorithms presented in papers [4]–[8] show the usage of the Iteration Space Slicing for coarse-grained parallelization. Coarse-grained parallelism is represented with synchronization-free slices or with slices requiring occasional synchronization. An (iteration-space) slice is defined as follows.

Definition 1: Given a dependence graph defined by a set of dependence relations, a *slice S* is a weakly connected component of this graph, i.e., a maximal subgraph such that for each pair of vertices in the subgraph there exists a directed or undirected path.

Definition 2: An *ultimate dependence source (resp. destination)* is a source (resp. destination) that is not the destination (resp. source) of another dependence. Ultimate dependence sources and destinations represented by relation R can be found by means of the following calculations: domain(R) − range(R) and range(R) − domain(R), respectively.

Definition 3: The set of ultimate dependence sources of a slice forms the set of its *sources*.

Definition 4: The *representative* source of a slice is its lexicographically minimal source.

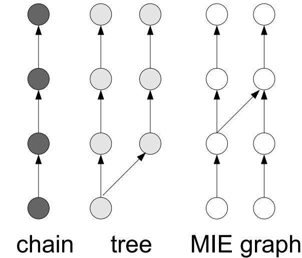

Fig. 1. Slices of the different topologies

We distinguish between the following topologies of slices

- *Single Incoming Edge* (SIE) graph, where each node has one and only one incoming edge. SIE graphs can represent the *chain* or *tree* topology.
- *Multiple Incoming Edges* (MIE) graph, where one or more nodes has multiple incoming edges.

Examples of slices with different topologies are shown in Figure 1.

Iteration Space Slicing algorithms extracting coarse-grained parallelism use standard operations on relations and sets, such as intersection (∩), union (∪), difference (−), domain of relation (domain(R)), range of relation (range(R)), identity relation (I), relation application (given a relation R and set S, $R(S) = \{[e'] :\ \exists e \in S, e \rightarrow e' \in R\}$), positive transitive closure

(given a relation R, $R^+ = \{[e] \to [e'] : e \to e' \in R \vee \exists e'', e \to e'' \in R^+ \wedge e'' \to e' \in R\}$), transitive closure ($R^* = R^+ \cup I$).

Notice that statement instances can be split into independent and dependent ones; extracting slices deals with dependent statement instances only: given a relation R capturing all dependences of a loop, the iteration space, S_{DEP}, that we consider is $S_{DEP} = \text{domain}(R) \cup \text{range}(R)$.

In order to scan independent statement instances, additional code should be generated as follows. We find set S_{IND} comprising independent statement instances as the difference between the space of all statement instances, S_{SI}, and the space of all dependent statement instances, S_{DEP},

$$S_{IND} = S_{SI} - S_{DEP}$$

and apply well-known code generation techniques [1], [2], [9], [20], [21] to generate code scanning elements of set S_{IND}.

Our approach to extract coarse-grained parallelism represented with synchronization-free slices [4]–[7] consists of the following steps

1) find set S_{repr} of representative sources of slices;
2) reconstruct slices from their representatives and generate code scanning these slices.

To facilitate exposition and implementation in Omega, we preprocess dependence relations to explicit source and destination statements and to normalize their tuples to the same dimension [5].

Given (preprocessed) dependence relation R representing all dependences in a loop, we can find a set of statement instances, S_{UDS}, describing all ultimate dependence sources as the difference between the domain of R and the range of R:

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

Next, we check whether R describes any common dependence destination, i.e., a statement instance that is a destination of two or more dependences. We find a set of common dependence destinations, S_{CDD}, as follows

$$S_{CDD} = \{[e] \quad : \quad e = R(e') = R(e'') \ \wedge \ e', e'' \in \text{domain}(R) \ \wedge \ e' \neq e''\}.$$

If set S_{CDD} is empty, we can conclude that all slices have either the chain or tree topology and each element of set S_{UDS} is the representative source of a slice, that is

$$S_{repr} = S_{UDS}.$$

Because R is an affine relation, S_{repr} being the difference between domain(R) and range(R) is also affine.

If order to discover the topology of slices, we find set of common dependence sources, S_{CDS},

$$S_{CDS} = \{[e] \quad : \quad e = R^{-1}(e') = R^{-1}(e'') \ \wedge \ e', e'' \in \text{range}(R) \ \wedge \ e' \neq e''\}.$$

If set S_{CDS} is empty, slices have the chain topology. Otherwise, slices have the tree topology.

When set S_{CDD} is not empty and slices have an MIE graph topology (that is neither a chain nor a tree), in order to find which elements of S_{UDS} are representatives of slices,[2] we build a relation R_{USC} that describes all pairs (e, e') of the ultimate dependence sources S_{UDS} that are transitively connected in a slice, i.e., $R^*(e) \cap R^*(e')$ is non-empty. Formally,

$$R_{USC} = \{[e] \to [e'] \quad : \quad e, e' \in S_{UDS} \ \wedge \ e \prec e' \wedge R^*(e) \cap R^*(e') \neq \varnothing\}.$$

It is evident that the set range(R_{USC}) contains all but the lexicographically minimal sources of synchronization-free slices. In order to find set S_{repr} of representatives of each slice (their lexicographically minimal statement instances), we have to perform the following computation:

$$S_{repr} = S_{UDS} - \text{range}(R_{USC}).$$

Note, that the computation of R_{USC} (and, respectively, S_{repr}) relies on the transitive closure of relation R, R^*, that has a closed form[3] and may be either affine or non-linear. If the closed form of R^* is affine, then the Omega Calculator [25] can be applied for computing R_{USC}. PIP/Piplib parametric integer linear programming solver [26] can also be used for this purpose. If the closed form of R^* is non-linear, tools like Mathematica [23] can be used to compute R_{USC}.

Having found set S_{repr} of representative sources of slices, we have to generate code scanning sources of synchronization-free slices and elements of each slice. Depending on the constraints of R^* and the topology of slices, the following algorithms can be applied for this purpose

- **Gen_affine** [4], [5] - an algorithm allowing us to generate code scanning synchronization-free slices of an arbitrary topology (chain, tree, MIE graph) if R^* has a closed form and affine constraints. This algorithm uses well-known code generation techniques to scan elements of affine sets representing synchronization-free slices [1], [2], [9], [20], [21].
- **Gen_chain** [6] - an algorithm allowing us to scan slices of the chain topology that does not require computation of R^*. It can be applied when **Gen_affine** fails to extract synchronization-free slices because of impossibility to compute affine R^* (which is the case for most non-uniform dependence loops).
- **Gen_tree** [7] - an algorithm allowing us to scan slices of the tree topology that does not require computation of R^*. Within each slice, it employs free-scheduling of statement instances (statement instances are executed as soon as their operands are available, preserving in such a way all dependences exposed for an original loop) and allows for enhancing code locality. It can be applied to parallelize loops for which **Gen_affine** fails to extract synchronization-free slices because of impossibility to

[2] If a slice has multiple sources, then although all its sources belong to S_{UDS}, only the lexicographically minimal source is the representative of a slice.

[3] A closed form expression is any formula that can be evaluated in a finite number of standard operations.

compute affine R^* (which is the case for most non-uniform dependence loops), or as an alternative for **Gen_affine** aiming at increasing code locality.

When the above algorithms extract only a single slice, the following algorithms can be used to seek for slices requiring synchronization

- **Synch_inLoop** [4], [5] - an algorithm searching for synchronization-free slices in inner loop nests. It generates code whose inner loops scan slices (when it is possible to extracts such slices by means of **Gen_affine**, **Gen_chain**, or **Gen_tree**) while outer loops are responsible for synchronization.
- **Synch_MP** [8] - an algorithm employing synchronization based on message passing using both OpenMP and POSIX locks functions. It is based on splitting a set of dependence relations into two sets. The first one is to be used for generating code scanning slices (applying algorithms **Gen_affine**, **Gen_chain**, or **Gen_tree**), while the second one permits us to insert send and receive functions to synchronize the slices execution.

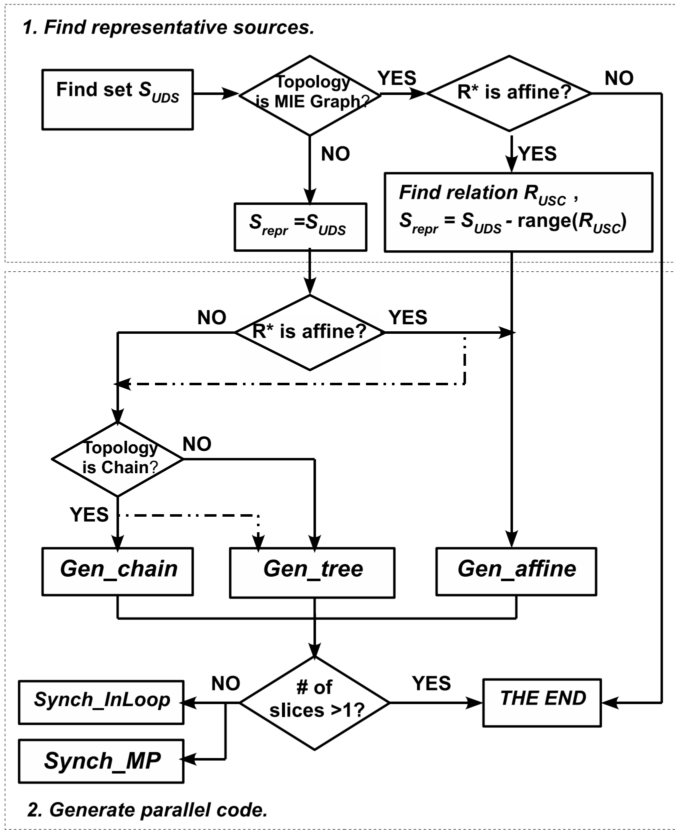

Fig. 2. Usage of ISS algorithms

Figure 2 shows the suggested scheme of the usage of the above coarse-grained parallelization algorithms. The continuous arrows show the suggested actions, while the dashed arrows show the alternative actions (such transitions are possible, but they are not necessarily efficient).

V. Iteration Space Slicing vs Affine Transformation Framework

Although the Affine Transformation Framework is known to be one of the most powerful techniques to extract parallelism in program loops, it has a list of limitations that does not permit for the extraction of available coarse-grained parallelism in loops. Below we present the limitation of ATF on various loops and show how Iteration Space Slicing algorithms can be applied to these loops to extract coarse-grained parallelism.

A. *Affine transformations fail to extract slices available in a single loop nest*

Example 1.

```
for (i=1;i<=n;i++)
  a(i)=a(i-3);
```

The dependence relation for this loop generated by Petit is the following

```
R:={[i] -> [i+3]: 1 <= i <= n-3}.
```

The dependences described by relation R for n=9 are shown in Figure 3, where different shades correspond to statement instances of different synchronization-free slices.

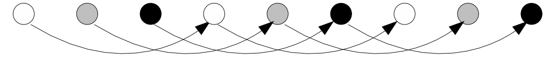

Fig. 3. Dependences for Example 1.

In order to find an affine transformation $\Phi = C_{11} \times i + c_1$ extracting coarse-grained parallelism for this example, we have to solve the following equation

$$C_{11} \times i + c_1 = C_{11} \times (i+3) + c_1$$

which we can simplify and rewrite as $3 \times C_{11} = 0$ and finally get $C_{11} = 0$. The solution [C_{11} c_1]=[0 c], for all positive integer c, means that there does not exist any affine transformation extracting two or more slices for this example.

Applying **Gen_affine** algorithm, we yield the following code scanning the three synchronization-free slices available in this loop.

```
(par)for(i = 1; i <= min(n-3,3); i++) {
  a(i)=a(i-3);
  for(j = i+3; j<= n; j+= 3)
    a(j)=a(j-3); }
```

where "(par)for" denotes a "for" loop whose iterations are independent and can be executed in parallel.

B. *Affine transformations are not able to extract all slices available in a loop*

Example 2.

```
for (i=1;i<=n;i++)
  for (j=1;j<=m;j++) {
s1:   a(i,j) = b(i,j)+c(i,j);
s2:   c(i,j-1) = a(i,j+1); }
```

The dependences originated by the loop above are described by the following dependence relations

```
R1:={s1[i,j]->s2[i,j+1]: 1<=i<=n & 1<=j<m};
R2:={s2[i,j]->s1[i,j+1]: 1<=i<=n & 1<=j<m}.
```

Figure 4 shows the dependences described with relations R1 and R2 for n=3 and m=6.

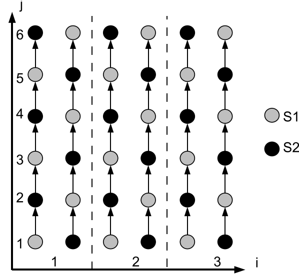

Fig. 4. Dependences for Example 2.

In order to find affine transformations extracting coarse-grained parallelism for this example, we have to find a solution to the following system of equations formed according to the dependence relations above

$$\begin{cases} C_{11}\times i+C_{12}\times j+c_1=C_{21}\times i+C_{22}\times (j+1)+c_2 \\ C_{21}\times i+C_{22}\times j+c_2=C_{11}\times i+C_{12}\times (j+1)+c_1 \end{cases}$$

The solution to the above system of equations is of the form [C_{11} C_{12} c_1]=[C_{21} C_{22} c_2]= [C 0 c], for any positive integer C, c. We can conclude that the general form of possible affine transformations for both statements s_1 and s_2 is $\Phi = n_1 \times i + n_2$, where $1 \leq i \leq n$, n_1 and n_2 are symbolic constants. According to this transformation, the instances of the same iteration of statements s_1 and s_2 are mapped to the same processor $n_1 \times i + n_2$. In other words, the number of independent threads is equal to n.

Applying algorithm **Gen_affine**, we generate two independent loops. Each loop scans n synchronization-free slices, so in total we obtain $2n$ slices, that is twice more than those found by ATF. The generated code is of the form

```
if (m >= 2) {
   (par)for(i = 1; i <= n; i++) {
      a(i,1) = b(i,1)+c(i,1);
      if (m >= 3 && i >= 1 && n >= i)
         c(i,1) = a(i,3);
      if (n >= i && i >= 1) {
         for(j = 3; j <= m; j++) {
            if (intMod(j+1,2) == 0)
               a(i,j) = b(i,j)+c(i,j);
            if (intMod(j,2) == 0)
               c(i,j-1) = a(i,j+1); }}
      if (m == 2 && i <= n && i >= 1)
         c(i,j-1) = a(i,j+1);}}
if (m >= 2) {
   (par)for(i = 1; i <= n; i++) {
      c(i,0) = a(i,2);
      if (m >= 3 && i >= 1 && n >= i)
         a(i,2) = b(i,2)+c(i,2);
      if (n >= i && i >= 1) {
         for(j = 3; j <= m; j++) {
            if (intMod(j,2) == 0)
               a(i,j) = b(i,j)+c(i,j);
            if (intMod(j+1,2) == 0)
               c(i,j-1) = a(i,j+1); }}
      if (m == 2 && i <= n && i >= 1)
         a(i,2) = b(i,2)+c(i,2); }}
```

C. Affine transformations do not permit for extraction of synchronization-free slices with multiple sources

Example 3.

```
for (i=1;i<=10;i++)
   for (j=1;j<=10;j++)
      a(j+4,j+1) = a(i+2*j+1,i+j+3)
```

The dependences originated by the loop above are described by the following dependence relations

```
R1:={[i,5]->[i,i+7]: 1 <= i <= 3};
R2:={[i,5]->[i',i+7]: 1<=i<i'<=10 && i<=3};
R3:={[1,9]->[2,5]} union {[2,10] -> [3,5]};
R4:={[i,j]->[j-7,5]: 1<=i<=j-8 && j<=10};
R5:={[i,j]->[i',j]: 1<=i<i'<=10 && 1<=j<=10}.
```

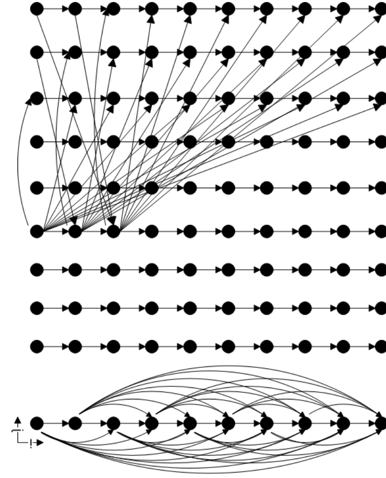

Fig. 5. Dependences for Example 3.

Figure 5 represents the dependences for this example. In order to improve the visualization of dependences, dependences described with relation R5 are fully shown only for i=1, while for the remaining values of i, $2 \leq i \leq 10$, only the dependences described as $[i, j]- > [i+1, j]$ are shown.

In order to find an affine transformation extracting coarse-grained parallelism for this example, we have to find a solution to the following system of equations formed according to the dependence relations above

$$\begin{cases} C_{11}\times i+C_{12}\times 5+c_1=C_{11}\times i+C_{12}\times (i+7)+c_1 \\ C_{11}\times i+C_{12}\times 5+c_1=C_{11}\times i'+C_{12}\times (i+7)+c_1 \\ C_{11}\times 1+C_{12}\times 9+c_1=C_{11}\times 2+C_{12}\times 5+c_1 \\ C_{11}\times 2+C_{12}\times 10+c_1=C_{11}\times 3+C_{12}\times 5+c_1 \\ C_{11}\times i+C_{12}\times j+c_1=C_{11}\times (j-7)+C_{12}\times 5+c_1 \\ C_{11}\times i+C_{12}\times j+c_1=C_{11}\times i'+C_{12}\times j+c_1 \end{cases}$$

There does not exist any non-zero solution to this system. Hence, there does not exist any affine transformation that extracts two or more synchronization-free slices available in this loop.

Applying algorithm **Gen_affine**, we are able to extract 7 synchronization-free slices for Example 3. The generated code is of the form

```
(par)for(j = 1; j <= 7; j++)
   for(i = 1; i <= 10; i++) {
      if (j <= 10 && i >= j-7 && j >= 9)
         a(j+4,j+1) = a(i+2*j+1,i+j+3);
      if (i <= 1)
         a(j+4,j+1) = a(i+2*j+1,i+j+3);
      for(j' = max(-i+j+2,j);
          j' <=min(intDiv(9*j-1,8),10); j'++)
            a(j+4,j+1)=a(i+2*j'+1,i+j'+3);
      if (j == 5) {
         for(j' = 8; j' <= 10; j'++)
            a(j+4,j+1)=a(i+2*j'+1,i+j'+3); } }
```

D. Affine transformations cannot extract synchronization-free parallelism available in a subspace of the loop domain

Analyzing the previous example, we can see that although there is no affine transformation exposing parallelism in the entire loops domain: {[i,j]: 1≤i,j≤10}, there exist synchronization-free slices (with a single source) in the iteration subdomain {[i,j]: 2≤j≤4, 1≤i≤10}. The code above generated by Algorithm **Gen_affine** represent all slices: with a single and multiple sources.

E. Affine transformations fail to extract coarse-grained parallelism from non-uniform loops exposing slices described with non-linear forms

Example 4.

```
for (i=1; i<=n; i++)
   for (j=1; j<=n; j++)

a(i,j)=a(2*i,1)-a(i,j+1);
```

Dependences in this loop are described with the following dependence relations

```
R1:={[i,1] -> [2i,1]: 1 <= i & 2i<=n};
R2:={[i,j] -> [i,j+1]: 1<=i<=n & 1<=j<n}.
```

Figure 6 presents the graphical representation of dependences for n=8 (different synchronization-free slices are shown in different shades).

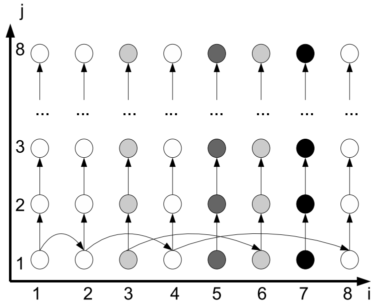

Fig. 6. Dependences for Example 4.

When we apply ATF to extract coarse-grained parallelism, we construct the following system of equations

$$\begin{cases} C_{11}\times i+C_{12}\times 1+c_1 = C_{11}\times 2i+C_{12}\times 1+c_1 \\ C_{11}\times i+C_{12}\times j+c_1 = C_{11}\times i+C_{12}\times (j+1)+c_1 \end{cases}$$

There does not exist any non-zero solution to this system. Hence, ATF fails to extract coarse-grained parallelism for this example.

Applying algorithm **Gen_trees**, we are able to extract $min(n,2*n-3)$ synchronization-free slices for this example. The generated code, containing fragments of pseudo-code in order to simplify the presentation, is as follows.

```
(par)for(i=1;i<=min(n,2*n-3);i+=2){
   I = [i, 1];
   Add I to S; /* I is a source of a tree */
   Exec(S); /* call function Exec() */
}

void Exec(S) {
   while(S!= 0) {
   S_tmp = 0;
   foreach(I in S) { /* I=[i,j] */
      a1[i][j] = a1[2*i][j]+a1[i][j+1];
        /* the original loop statement to be
           executed at iteration I=(i,j)*/
      if(j == 1 && 1 <= i && 2*i <= n) {
         /* if R1(I) is in domain(R1) */
         ip = 2*i; jp = 1;
            /* J=[ip,jp]=R1(I) */
         add J to S_tmp; }
      if(1<=i && i<=n && 1<=j && j<n){
         /* if R2(I) is in domain(R2) */
         ip=i; jp = 1 + j;
            /* J=[ip,jp]=R2(I) */
         Add J to S_tmp; } }
   S = S_tmp; } }
```

F. Affine transformations cannot be applied to extract non-synchronization-free slices

Example 5.

```
for(i = 1; i <= n; i++)
   for(j = 1; j <= n; j++)
      a(i,j)=a(i-1,j)+a(i,j-3)+a(i-1,j-1);
```

Petit extracts the following dependence relations for this loop

```
R1:={[i,j]->[i+1,j]: 1<=i<n & 1<=j<=n};
R2:={[i,j]->[i,j+3]: 1<=i<=n & 1<=j<=n-3};
R3:={[i,j]->[i+1,j+1]: 1<=i<n & 1<=j<n}.
```

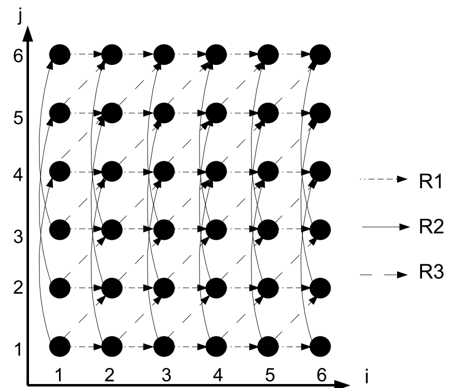

Fig. 7. Dependences for Example 5.

Figure 7 illustrates dependences described with the above relations for n=6. When we are looking for an affine transformation $\Phi = [C_{11} C_{12}] \times i + c_1$ for this example, we form and resolve the following system of equations

$$\begin{cases} C_{11} \times i + C_{12} \times j + c_1 = C_{11} \times (i+1) + C_{12} \times j + c_1 \\ C_{11} \times i + C_{12} \times j + c_1 = C_{11} \times i + C_{12} \times (j+3) + c_1 \\ C_{11} \times i + C_{12} \times j + c_1 = C_{11} \times (i+1) + C_{12} \times (j+1) + c_1 \end{cases}$$

There is no non-zero solution to the system above. Hence, ATF does not expose any transformation extracting coarse-grained parallelism for Example 5.

Applying algorithm **Synch_inLoop** (which uses algorithm **Gen_affine**), we can extract the three available slices in the inner loop nest. The code scanning these slices is of the form

```
for (i=1; i<=n; i++)
   (par)for(j' = 1; j' <= min(n-3,3); j'++)
      for(j = j'; j <= n; j += 3)
         a(i,j)=a(i-1,j)+a(i,j-3)+a(i-1,j-1);
```

VI. Need for advanced techniques for calculating transitive closure

The discussed Iteration Space Slicing algorithms were implemented by us using the Omega library [14], [25] in a tool permitting for automatic calculating sources of slices, recognizing topologies of slices, and generating C-like pseudo-code scanning either synchronization-free slices or slices requiring synchronization. Applying this tool, we have experimented with loops of the NAS 3.2 [24] and UTDSP [27] benchmarks to recognize how many loops can be parallelized with the ISS algorithms and what is the topology and the number of slices extracted for those loops.

We have studied only those loops for which Omega's dependence analyzer — Petit [25] — was able to carry out exact dependence analysis (Petit fails to analyze loops containing the "break", "goto", "continue", "exit" statements, functions and when array indexies are elements of other arrays), and where there exists at least one data flow dependence (anti and output dependences in ISS computations can be eliminated by means of well-known techniques). We have chosen Petit because it implements the exact dependence analysis proposed by Pugh and Wonnacott [19] and produces dependence relations defined by Presburger formulas. We are not aware of any other publically available tool performing the exact dependence analysis. From 431 loops of the NAS benchmark, Petit was able to extract dependences from 257 loops and data flow dependences were present in 149 loops. From 78 loops of the UTDSP benchmark, Petit was able to extract dependences from 43 loops and flow dependences were present in 34 UTDSP loops.

For 149 NAS loops and 34 UTDSP loops qualified to experiments, the tool extracts slices for 96 NAS and 18 UTDSP loops. For 5 NAS and 5 UTDSP benchmarks only a single slice can be extracted, while for 48 NAS loops and 11 UTDPS loops (which is around 32% of considered benchmarks), the tool is not able to extract slices because Omega fails to calculate exact transitive closure required for extracting sources of slices for those loops. It is worth to note that this is not a feature of the algorithms but that of the tool. In the matter of fact, the heuristic approaches proposed in [15] and implemented in Omega [25] do not guarantee the calculation of exact transitive closure and often fail to produce any result even for loops with simple uniform dependences.

Table I presents the number of loops for which coarse-grained parallelism was extracted, the number and percentage of loops of the chain, tree, and MIE graph topologies. Table II presents the summary on the usage of algorithms **Gen_affine**, **Gen_chain**, **Gen_tree**, **Synch_InLoop** and **Synch_MP** and expressed in the number of loops from the NAS and UTDSP benchmarks from which particular algorithms were able to extract coarse-grained parallelism. The notations 3(+32) and 1(+3) in the column for **Gen_tree** algorithm mean that this algorithm extracts synchronization-free trees from 3 NAS and 1 UTDSP loops and it can be also applied to extract synchronization-free chains from other 32 NAS and 3 UTDSP loops (though in this case the generated code will be less efficient than that generated by **Gen_chain**, because slices have the chain topology and no additional synchronization implied by free scheduling is needed). The numbers concerning algorithms **Synch_InLoop** and **Synch_MP** show how many loops it was possible to parallelize **only** introducing synchronization (in the matter of fact, those algorithms can be applied to parallelize most NAS and UTDSP benchmarks, but because we are interested in extraction of maximal coarse-grained parallelism, we apply them only if **Gen_affine**, **Gen_chain** and **Gen_tree** algorithms fail to extract two or more slices).

Carried out experiments demonstrate that there exists a strong need for devising novel techniques for calculating exact transitive closure. In our opinion, this will permit for producing not only acadameic, but also industrial compilers based on the ISS framework.

VII. Conclusion and Future Work

In this paper we demonstarted that ISS extracts more coarse-grained parallelism than that exstrated by ATF. In paper [18], Pugh and Rosser give the explanation of this fact: "ISS is a finer-grained approach than many existing transformations, which might act on all statements in a loop, or all iterations of an individual statement. A slice includes all and only those statement instances which must be executed to produce the correct output. It is a data-driven technique in that an optimizer does not work with the loop structure, but instead specifies which data should be computed".

Currently we carry out research in the following complementary directions to strengthen the power of ISS

1) Advanced techniques for calculating the exact transitive closure of a union of affine relations. Although the symbolic computation of the exact transitive closure of a parameterized affine relation is not possible in general [15], it is possible to define special cases for which symbolic computation of R^* can be done.
2) Approximations of R^* when the exact R^* cannot be computed symbolically. Only very coarse approximations have been proposed so far, especially in the (common) case of non-convex iteration sets. Approximations will

Benchmark	# of Loops	Topology					
		Chain		Tree		MIE Graph	
NAS	96	32	33.3%	4	4.2%	60	62.5%
UTDSP	18	3	16.7%	1	5.6%	14	77.7%

TABLE I
RESULTS OF EXPERIMENTS

Benchmark	Algorithms				
	Gen_affine	Gen_chain	Gen_tree	Synch_InLoop	Synch_MP
NAS	63	32	3 (+32)	6	3
UTDSP	17	3	1 (+3)	0	2

TABLE II
USAGE OF THE ALGORITHMS

lead to suboptimal extraction of parallelism; the induced loss of scalability will need to be investigated.

3) Approaches to extract synchronization-free slices that have an MIE graph topology (that is neither a chain nor a tree) and are described with non-linear forms.

REFERENCES

[1] Corinne Ancourt and François Irigoin. Scanning polyhedra with do loops. In *PPOPP'91: Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 39–50, New York, NY, USA, 1991. ACM.

[2] Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 7–16, Juan-les-Pins, september 2004.

[3] Cédric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma, A Group, and Inria Rocquencourt. Putting polyhedral loop transformations to work. In *In Workshop on Languages and Compilers for Parallel Computing (LCPC03), LNCS*, pages 209–225. Springer Verlag, 2003.

[4] Anna Beletska, Wlodzimierz Bielecki, and Pierluigi San Pietro. Extracting coarse-grained parallelism in program loops with the slicing framework. In *ISPDC '07: Proceedings of the Sixth International Symposium on Parallel and Distributed Computing*, page 29, Washington, DC, USA, 2007. IEEE Computer Society.

[5] Anna Beletska, Wlodzimierz Bielecki, Krzysztof Siedlecki, and Pierluigi San Pietro. Finding synchronization-free slices of operations in arbitrarily nested loops. In *ICCSA (2)*, volume 5073 of *Lecture Notes in Computer Science*, pages 871–886. Springer, 2008.

[6] Wlodzimierz Bielecki, Anna Beletska, Marek Palkowski, and Pierluigi San Pietro. Extracting synchronization-free chains of dependent iterations in non-uniform loops. In *ACS '07: Proceedings of International Conference on Advanced Computer Systems*, 2007.

[7] Wlodzimierz Bielecki, Anna Beletska, Marek Palkowski, and Pierluigi San Pietro. Finding synchronization-free parallelism represented with trees of dependent operations. In Anu G. Bourgeois and Si-Qing Zheng, editors, *ICA3PP*, volume 5022 of *Lecture Notes in Computer Science*, pages 185–195. Springer, 2008.

[8] Wlodzimierz Bielecki and Marek Palkowski. Using message passing for developing coarse-grained applications in OpenMP. In *ICSOFT '08: Proceedings of International Conference on Software and Data Technologies*, 2008.

[9] Pierre Boulet, Alain Darte, Georges-André Silber, and Frédéric Vivien. Loop parallelization algorithms: from parallelism extraction to code generation. *Parallel Comput.*, 24(3-4):421–444, 1998.

[10] Alain Darte, Yves Robert, and Frederic Vivien. *Scheduling and Automatic Parallelization*. Birkhaser, 2000.

[11] Paul Feautrier. Some efficient solutions to the affine scheduling problem: I. one-dimensional time. *Int. J. Parallel Program.*, 21(5):313–348, 1992.

[12] Paul Feautrier. Some efficient solutions to the affine scheduling problem: Ii. aaamulti-dimensional time. *Int. J. Parallel Program.*, 21(5):389–420, 1992.

[13] Paul Feautrier. Toward automatic distribution. *Parallel Processing Letters*, 4, 1994.

[14] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott. The omega library interface guide. Technical report, College Park, MD, USA, 1995.

[15] Wayne Kelly, William Pugh, Evan Rosser, and Tatiana Shpeisman. Transitive closure of infinite graphs and its applications. *Int. J. Parallel Programming*, 24(6):579–598, 1996.

[16] Amy W. Lim, Gerald I. Cheong, and Monica S. Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. In *In Proceedings of the 13th ACM SIGARCH International Conference on Supercomputing*, pages 228–237. ACM Press, 1999.

[17] Amy W. Lim and Monica S. Lam. Communication-free parallelization via affine transformations. In *In 24 th ACM Symp. on Principles of Programming Languages*, pages 92–106. Springer-Verlag, 1994.

[18] William Pugh and Evan Rosser. Iteration space slicing and its application to communication optimization. In *International Conference on Supercomputing*, pages 221–228, 1997.

[19] William Pugh and David Wonnacott. An exact method for analysis of value-based array data dependences. In *In Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing*. Springer-Verlag, 1993.

[20] Fabien Quilleré, Sanjay Rajopadhye, and Doran Wilde. Generation of efficient nested loops from polyhedra. *Int. J. Parallel Program.*, 28(5):469–498, 2000.

[21] Nicolas Vasilache, Cédric Bastoul, and Albert Cohen. Polyhedral code generation in the real world. In *Proceedings of the International Conference on Compiler Construction (ETAPS CC'06)*, LNCS 3923, pages 185–201, Vienna, Austria, March 2006. Springer-Verlag.

[22] Mark Weiser. Program slicing. In *IEEE Transactions on Software Engineering*, pages 352–357, 1984.

[23] Wolfram Mathematica. http://www.wolfram.com/products/mathematica/index.html.

[24] NAS benchmarks suite. http://www.nas.nasa.gov.

[25] The Omega project. http://www.cs.umd.edu/projects/omega.

[26] Piplib - a parametric integer linear programming solver. http://www.prism.uvsq.fr/~cedb/bastools/piplib.html.

[27] UTDSP benchmarks suite. http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html.