Synchronization-Free Automatic Parallelization: Beyond Affine Iteration-Space Slicing

Anna Beletska, Wlodzirmierz Bielecki, Albert Cohen, Palkowski Marek

Synchronization-Free Automatic Parallelization: Beyond Affine Iteration-Space Slicing

Anna Beletska[1], Wlodzimierz Bielecki[2], Albert Cohen[3], and Marek Palkowski[4]

[1] INRIA Saclay, France, Anna.Beletska@inria.fr
[2] West-Pomeranian Technical University, Poland, WBielecki@wi.ps.pl
[3] INRIA Saclay, France, Albert.Cohen@inria.fr
[4] West-Pomeranian Technical University, Poland, MPalkowski@wi.ps.pl

Abstract. *This paper contributes to the theory and practice of automatic extraction of synchronization-free parallelism in nested loops. It extends the iteration-space slicing framework to extract slices described by not only affine (linear) but also non-affine forms. A slice is represented by a set of dependent loop statement instances (iterations) forming an arbitrary graph topology. The algorithm generates an outer loop to spawn synchronization-free slices to be executed in parallel, enclosing sequential loops iterating over those slices. Experimental results demonstrate that the generated code is competitive with that generated by state-of-the-art techniques scanning polyhedra.*

1 Introduction and Related Work

Programming multicore systems is a very challenging process aimed at taking full advantage of the computing potential available in multicore-based computers. Parallelizing compilers extract parallelism automatically from existing sequential applications. In this paper, we deal with synchronization-free parallelization obtained by creating a thread of computations on each processor to be executed independently. This task is critical for multiprocessors with shared memory as it allows us to compensate for the overhead caused by exploiting parallelism and process synchronization.

Pugh and Rosser [29] introduced *iteration-space slicing*, an extension of (scalar) program slicing specialized for nested loops, to compute the *precise set of statement instances* (iterations) associated with the definition of a given *array element*. This technique has obvious applications to synchronization-free parallelization: Beletska et al. [4, 5] demonstrated how to extract synchronization-free threads described only by affine forms.

According to Yu et at. [34], 46% of the nested loops in the SPECfp95 benchmark contain non-uniform data dependences. Furthermore, coupled array subscripts, i.e., indices that appear in both dimensions, often cause non-uniform dependences as well. A study of 12 other benchmarks shows that about 45% of two dimensional array reference pairs are coupled linear subscripts. Including one-dimensional arrays, about 12.8% of the coupled subscripts in the SPECfp95 benchmarks generate non-uniform dependences. Examples of such programs containing non-uniform dependences can be found in Linpack, Eispack, Itpack, and Fishpak benchmarks [26], ADI and SYR2K benchmarks [27].

Non-uniform dependences can result in slices being described not only by affine forms but also by non-affine ones, e.g., by non-linear expressions.

Yu et al. [34] present an approach tackling non-uniform loops and permitting for scanning chains only; the approach does not permit for extracting synchronization-free slices of an arbitrary topology.

The *affine transformation framework* (ATF) [13, 14, 18, 22] unifies a large class of loop transformations and is considered one of the most powerful frameworks to extract both coarse- and fine-grained parallelism today. But it is associated with significant limitations restricting its expressiveness and its ability to extract synchronization-free parallelism in a significant proportion of loop nests [6]. In particular, it fails to extract slices in most practical non-uniform loops and cannot find slices that are described with non-affine expressions. The *iteration-space slicing* framework is more expressive in terms of parallelism extraction as it is not restricted to affine partitions.

Iteration-space slicing techniques proposed by Beletska et al. in [4, 5] are restricted to the extraction of synchronization-free parallelism that can be represented as unions of affine expressions; this is mostly due to the limitations of state-of-the-art code generation algorithms [1, 2, 10, 19, 30, 32]. Although those algorithms can successfully parallelize many real-world loops, they cannot extract synchronization-free parallelism available in non-uniform loops and being presented by slices described with non-affine forms. Bielecki et al. [7] showed how to deal with such slices and how to generate parallel code when dependences form the chain or the tree topology. In the current paper, we extend those results to the most general case of imperfectly nested loops with (affine) *dependences forming an arbitrary graph topology*. We present a technique that permits to extract synchronization-free slices described by not only affine but also *non-affine* forms and to generate efficient parallel code scanning slices.

2 Background

In this paper, we 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. We consider arbitrary (imperfect) loop nestings. A *statement instance* is a particular execution of a statement of the loop body. A statement instance $S(I)$ is formed of the statement S itself and its *iteration vector* I, composed of the values of the surrounding loop indices.

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 $S_2(J)$, they are respectively called the *source* and *destination* of the dependence. The sequential execution ordering of statement instances, denoted as $S_2(J) \prec S_2(J)$, is induced by the lexicographic ordering of iteration vectors[5] and the textual ordering of statements when the instances share the same iteration vector.

Our approach to extract parallelism assumes an exact representation of loop-carried dependences and consequently an exact dependence analysis which detects a depen-

[5] Lexicographic ordering is an ordering for the Cartesian product (denoted as $\times$) of any two sets A and B with order relations $< A$ and $< B$, respectively, such that if (a_1,b_1) and (a_2,b_2) both belong to A$\times$B, then $(a_1,b_1)<(a_2,b_2)$ iff either $(a_1 < Aa_2)$ or $(a_1 = a_2$ and $b_1 < Bb_2)$.

dence if and only if it actually exists at runtime between two given instances.[6] To describe and implement our algorithms at a high level, we represent dependences by relations whose constraints are described in Presburger arithmetic (built of affine expressions, logical and existential operators); we use the Omega calculator for computations over such relations [24].

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

$$\{[input\_list] \rightarrow [output\_list] \mid constraints\},$$

where *input_list* and *output_list* are 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*. We use standard operations on relations and sets, such as intersection ($\cap$), union ($\cup$), difference ($-$), domain of relation (domain(R)), range of relation (range(R)), identity relation (I), relation application (given a relation R and set S, $R(S) = \{[e'] \mid \exists e \in S, e \rightarrow e' \in R\}$), positive transitive closure (given a relation R, $R^+ = \{[e] \rightarrow [e'] \mid e \rightarrow e' \in R \vee \exists e'', e \rightarrow e'' \in R^+ \wedge e'' \rightarrow e' \in R\}$), transitive closure ($R^* = R^+ \cup I$).

Iteration-space slicing [29] takes dependence information as input to find all statement instances that must be executed to produce the correct values for the specified array elements. In this paper, a dependence graph always refers to the extensive set of dependences of a loop nest, described by dependence relations in Presburger arithmetic. We define an (iteration-space) slice as follows.

Definition 1. *Given a (possibly unbounded/parameterized) 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.*

If there exist two or more slices in a dependence graph, the above definition guarantees that all these slices are *synchronization-free* when executed as concurrent threads (there is no dependence between them).

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*[7].

The set of ultimate dependence source(s) of a slice forms the so-called *source(s)* of the slice.

The topology of a slice can be a chain, tree, or arbitrary graph (neither tree nor chain). Examples of slices of the different topologies are shown in Figure 1.

[6] 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.

[7] Parametric Integer Programming (PIP) [25] allows to compute these sets very efficienty.

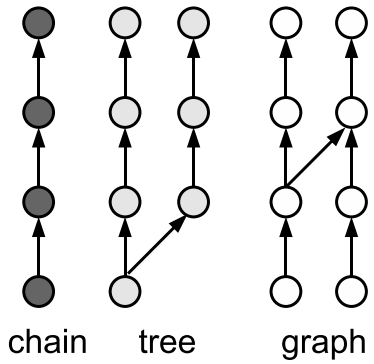

Fig. 1. Slices of the different topologies

In this paper, we present an algorithm to extract synchro-nization-free slices of an arbitrary graph topology and generate parallel code scanning such slices.

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 nest, the iteration space that we consider is domain$(R) \cup$ range(R).

3 Motivating Example

In particular, non-uniform dependences occur when array subscripts contain expressions of the form $a \times i$, where $a > 1$ and i is a loop index variable. According to Shen et al. [31], such expressions are present in 9% of single-nest loops, 2% of double-nested loops, and 16% of triple-nested loops from the studied by authors 12 benchmark suites. Let us consider the loop nest in Figure 2 containing the expression `2*i` in the array subscripts.

```
for (i=1; i<=n; i++)
  for (j=1; j<=m; j++)
    a[i][j] = a[2*i][j] + a[i][j-1];
```

Fig. 2. Motivating example

This loop is associated with the following non-uniform dependence relation (see Figure 3 for the graphical representation for $n = 8$ and $m = 4$):

$$R = \{[i, j] \rightarrow [2i, j] \mid 2 \leq 2i \leq n \wedge 1 \leq j \leq m\} \cup \\ \{[i, j] \rightarrow [i, j+1] \mid 1 \leq i \leq n \wedge 1 \leq j < m\}.$$

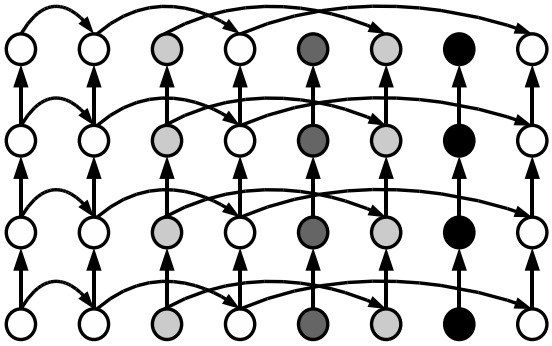

Fig. 3. Dependences in the motivating example

From Figure 3, we observe that for $n = 8$ and $m = 4$ there exist 4 synchronization free slices (statement instances of different slices are printed with different colors). Increasing the value of n results in increasing the number of slices: $\lceil n/2 \rceil$ in general.

In order to find an affine transformation extracting synchro-nization-free parallelism for this example, the model proposed by Lim et al. [22] applied to the above relations yields the following system of equations:

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

The solutions to this system for all i, j are of the form $[C_{11}, C_{12}, c_1] = [0, 0, c]$ for all non-negative integer c. Such a solution means that the affine transformation framework fails to extract two or more slices for this example because slices are described by non-affine forms. We are unaware of any other technique allowing us to extract synchronization-free parallelism available in this loop nest.

In the following sections we show how to extract and generate code for all slices available in this example.

4 Extracting synchronization-free slices

Our approach to extract synchronization-free slices takes two steps. First, for each slice, a representative statement instance is defined (it is the lexicographically minimal statement instance – one of the sources – of the slice). Next, slices are reconstructed from their representatives and code scanning these slices is generated.

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

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

In order to find which elements of S_{UDS} are representatives of slices[8], 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,

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

The set domain(R_{USC}) contains all but the lexicographically maximal sources of slices with multiple sources, while the set range(R_{USC}) contains all but the lexicographically minimal sources of such slices. In order to find set S_{repr} of representatives of each slice (their lexicographically minimal statement instances [25]), we have to perform the following calculation:

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

For the motivating example:

$$\begin{aligned}
R^* &= \{[i,j] \rightarrow [2^{k_1} \times i, j+k_2] \mid \exists k_1,k_2 \geq 1 \wedge 1 \leq i \wedge 2^{k_1} \times i \leq n \\
&\qquad \wedge 1 \leq j \wedge j+k_2 \leq m\} \cup \{[i,j] \rightarrow [i,j] \mid 1 \leq i \leq n \wedge 1 \leq j \leq m\} \\
S_{UDS} &= \{[i,1] \mid \exists \alpha, 2\alpha = i+1 \wedge 1 \leq i \leq \min(n, 2n-3)\} \\
R_{USC} &= \varnothing \\
S_{repr} &= S_{UDS}
\end{aligned}$$

Notice the exponential (hence non-affine) symbolic bound $2^{k_1} \times i$. Since there does not exist a general algorithm to compute the transitive closure of an affine relation, some of the above computations may not always be tractable. In the following, we only consider dependence relations where a closed form for R^*, R_{USC} and S_{repr} can be computed. Nevertheless, as illustrated on the motivating example, *such closed form expressions may not necessarily be affine*.

Practical means to compute non-affine transitive closures have been proposed for the sake of induction variable analysis and classification [33, 12]. Beyond compilation purposes, it is related with decision procedures in formal verification (through timed automata and hybrid automata [16]) and can also be computed with formal calculus tools (Maple, Mathematica, etc.); see the work of Boigelot and Wolper [9], Comon and Jurski [11].

[8] 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.

4.1 Code generation for a general graph topology

Input: relation R representing all dependences in a loop nest; set S_{repr} of representatives of synchronization-free slices; relation R_{USC} describing sources of slices that are connected.
Output: code scanning synchronization-free slices preserving all dependences

```
// Generate code to scan slice representatives and to spawn synchronization-free threads
gen_code(S_repr) {
  NEXT_repr = {[e] → [e'] | e,e' ∈ S_repr ∧ e ≺ e' ∧ ∄e'' ∈ S_repr, e ≺ e'' ≺ e'};
  I = domain(NEXT_repr) − range(NEXT_repr)
  // Generate inner loop to scan slice representative
  do {
    // Create a thread to run the slice represented by I
    asynchronous_spawn(gen_slice(I));
    // Update I to define the next slice representative
    I = NEXT_repr(I);
  } while (I∈ domain(NEXT_repr));
  asynchronous_spawn(gen_slice(I));
}

// Scan statement instances in the slice represented by I
gen_slice(I) {
  // Compute the set of all statement instances in the slice
  S_inst = (R* ∘ R_USC*)(I);
  NEXT_inst = {[e] → [e'] | e,e' ∈ S_inst ∧ e ≺ e' ∧ ∄e'' ∈ S_inst, e ≺ e'' ≺ e' ∧ e''};
  // Generate inner loop to scan statement instances while satisfying dependences
  do {
    // Run the loop statement for instance I
    execute(I);
    // Update I to define the next statement instance
    I = NEXT_inst(I);
  } while (I∈ domain(NEXT_inst));
  // Run the loop statement instance I
  execute(I);
}
```

Fig. 4. Code generation based on forming a single dependence relation

Figure 4 presents our algorithm to generate code scanning synchronization-free slices that are described with general non-affine forms, and have an arbitrary graph topology.

Important syntactic convention: the underlined font denotes code to be emitted by the algorithm, while the normal font denotes static computations to emit the proper code.

The main idea of this algorithm is to build a "next representative" relation $NEXT_{repr}$ and a "next instance" relation $NEXT_{inst}$ to iterate through sets of instances described

by non-affine constraints. As an additional design property, the "next instance" relation $NEXT_{inst}$ walks through each slice while preserving all dependences of the directed acyclic graph. The most canonical way consists in following the sequential execution order of the source program; this order is a total (chain topology) super-order of the partial dependence order. We define these relations as follows:

$$NEXT_{repr} = \{[e] \to [e'] \mid e, e' \in S_{repr} \wedge e \prec e' \wedge \nexists e'' \in S_{repr}, e \prec e'' \prec e'\}$$

$$NEXT_{inst} = \{[e] \to [e'] \mid e, e' \in S_{inst} \wedge e \prec e' \wedge \nexists e'' \in S_{inst}, e \prec e'' \prec e'\}$$

Based on these relations, it is possible to spawn concurrent threads for all slices and to generate relatively simple and efficient code scanning elements of those slices. This idea is the reminiscent of a similar approach to scanning static-control loop nests by Boulet and Feautrier [15]; they used parametric integer linear programming to compute a "next instance" relation over unions of polyhedra.

So far, we relied on the computation of closed form expressions for S_{repr} and R_{USC}, assuming the effective computability of the transitive closure of the dependence relation. We also assume that closed form expressions exist for $NEXT_{repr}$ and $NEXT_{inst}$, since the "next representative" and "next instance" relations can be obtained from simple operations like composition, intersection and difference. Indeed, let $(\prec)$ denote the relation capturing the sequential execution order, one may define a relation R_{repr} (resp. R_{inst}) mapping representatives (resp. instances) into the subsequent ones, and compute the "next representative" (resp. "next instance") relation as follows:

$$\begin{aligned} R_{repr} &= (S_{repr} \times S_{repr}) \cap (\prec) \\ S_{inst} &= R^* \circ {R_{USC}}^*(I) \\ R_{inst} &= (S_{inst} \times S_{inst}) \cap (\prec) \\ NEXT_{repr} &= R_{repr} - R_{repr} \circ R_{repr} \\ NEXT_{inst} &= R_{inst} - R_{inst} \circ R_{inst} \end{aligned}$$

Assuming intersection with $(\prec)$, composition and difference result in closed form expressions for S_{repr} and S_{inst}, these equations allow us to derive closed form expressions for $NEXT_{repr}$ and $NEXT_{inst}$. In Section 4.3 we study practical classes of non-affine closed forms that support these composition, intersection, and difference operations. The intuition is that if R^* can be computed, it is very likely that these operations will be computed as well.

On the other hand, this algorithm can be enhanced to detect when S_{repr} or R_{USC} happen to be constrained by affine forms only, triggering more efficient polyhedral scanning techniques [1, 19, 10, 30, 2, 32].

4.2 Illustration on the motivating example

Let us illustrate the code generation algorithm in Figure 4 using the motivating example of Figure 2.

Figure 5 shows the sequential execution order: elements of each synchronization-free slice are scanned according to the algorithm in Figure 4.

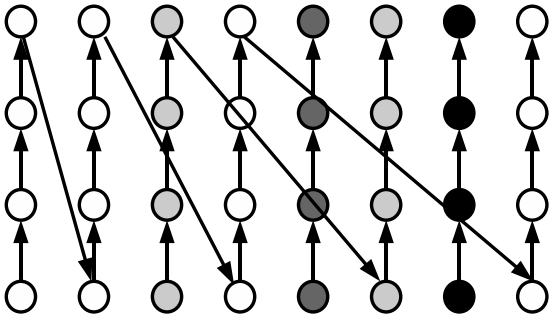

Fig. 5. Scanning iterations in parallel slices

The first step consists in emitting outer loops to scan sources of slices. Because set S_{repr} is defined with affine constraints, this can be achieved by means of Omega's code generator "codegen" [19, 24]; therefore, `asynchronous_spawn()` can be substituted with a `forall` loop.

```
forall (t=1; t<=min(n,2*n-3); t+=2) {
  // I = [i,j] = [t,1]
  gen_slice(t,1);
}
```

Then, we compute
$NEXT_{inst} = \{[i,j] \rightarrow [i,j+1] \mid 1 \leq i \leq n \wedge 1 \leq j < m\} \cup \{[i,j] \rightarrow [2i,1] \mid 1 \leq i \wedge 2i \leq n \wedge j = m\}$
and generate code that can be represented with the pseudo-code in Figure 6.

```
forall (t=1; t<=min(n, 2*n-3); t+=2) {
  [i,j]=[t,1]; // I = I = [i,j] is a representative source of each slice
  do {
    a[i][j] = a[2*i][j] + a[i][j-1];
    if (1<=i<=n && 1<=j<m) {
      // If I belongs to domain(NEXT_inst), as implied by the first disjunct of NEXT_inst
      j=j+1; // I = [i,j] = NEXT_inst(I)
    } else if (1<=i && 2*i<=n && j==m) {
      // If I belongs to domain(NEXT_inst), as implied by the second disjunct of NEXT_inst
      i=i*2; j=1; // I = [i,j] = NEXT_inst(I)
    }
  } while (1<=i<=n && 1<=j<m || 1<=i && 2*i<=n && j==m);
  a[i][j] = a[2*i][j] + a[i][j-1];
}
```

Fig. 6. Parallel pseudo-code for the motivating example

4.3 Approach applicability

The proposed approach allows us to parallelize not only uniform but also non-uniform loops. We have already mentioned that the two following kinds of array subscripts cause non-uniform data dependences

1. $a \times i_k$, where $a > 1$ is the integer constant and i_k, $k \in \{1, 2, ..., n\}$, is the loop index variable.
2. coupled subscripts $a_1 \times i_1 + a_2 \times i_2 + ... + a_n \times i_n$, where $a_1, a_2, ..., a_n \geq 0$ are the integer constants at least two of which are greater than or equal to 1, and $i_1, i_2, ..., i_n$ are the loop index variables.

Dependence relations caused by the both kinds of array subscripts can be illustrated with the following relations R_1 and R_2, respectively

$R_1 = \{[i] \rightarrow [a \times i] \mid L \leq i, a \times i \leq U\}$,

$R_2 = \{[i_1, i_2] \rightarrow [i_1, i_1 + i_2] \mid L \leq i_1, i_2, i_1 + i_2 \leq U\}$,

where L and U are the symbolic constants defining the domain and range of relations R_1 and R_2.

Our approach to extract synchronization-free slices described with non-affine forms requires transitive closure of non-uniform dependence relations.

Bielecki et al. in [8] show how to compute the transitive closure of a single dependence relation caused by array subscripts of both the first and second kinds. The exact transitive closure calculation is based on resolving a system of recurrence equations being formed from the input and output tuples of a dependence relation. Applying that technique, the transitive closure of relation R_1 is of the form

$R_1^* = \{[i] \rightarrow [a^k \times i] \mid \exists k : L \leq i, a^k \times i \leq U\}$

while that of relation R_2 is the following

$R_2^* = \{[i_1, i_2] \rightarrow [i_1, k \times i_1 + i_2] \mid \exists k : L \leq i_1, i_2, k \times i_1 + i_2 \leq U\}$.

R_1^* is a non-affine relation, but because of the properties of the exponential function, it clearly defines a class of relations that is closed for the operations we need to generate code ($\circ$, $\cap$, -).

Dealing with unions involving relations of both kinds is much more difficult – there is no known closed-form expression for the transitive closure of a union of affine dependence relations in general. Nevertheless, Beletska et al. in [3] propose an approach to compute the transitive closure of a union of affine relations, assuming that transitive closure of each single relation can be computed. They introduce a sufficient and necessary condition defining a class of relations for which the exact computation is possible.

Relation R in the motivating example is the union of two relations that satisfy the sufficient and necessary condition presented in [3]. Thus, approach described in [3] can be applied to compute R^*. Again, since the exponential and affine components of R^* occupy orthogonal subspaces, this relations also belongs to a class supporting our three operations ($\circ$, $\cap$, -).

5 Experiments

In order to study the performance of programs generated by the proposed algorithm, we produced parallel code in OpenMP for the motivating example as well as 8 computationally heavy loop nests from the NAS benchmarks[23].

We measured the execution time of the parallel programs on 1, 2, 4 and 8 processors in the following environment: Intel Xeon 1.6 Ghz, 8 processors (two quad-core processor, 4MB cache), 2 GB RAM, Ubuntu Linux, showing both speedup (ith respect to original sequential code) and efficiency numbers.

Table 1 presents the results for the motivating example on 1, 2, 4, and 8 processors: column "*N*" shows the value of the upper bounds of the loop indices, column "original" shows the execution time (in seconds) of the original `for` loop on 1 processor, column "`while`" shows the execution time of the generated `while`-based code on 1 processor, columns "time[s]", "S" and "E" show the time of the execution, speedup, and efficiency of the generated parallel code on 2, 4, and 8 processors.

	1 CPU		2 CPU			4 CPU			8 CPU		
N	original	`while`	time[s]	S	E	time[s]	S	E	time[s]	S	E
1000	0.024	0.024	0.016	1.486	0.743	0.014	1.727	0.432	0.012	1.932	0.242
1500	0.054	0.054	0.033	1.617	0.808	0.026	2.069	0.517	0.022	2.423	0.303
2000	0.096	0.096	0.052	1.844	0.922	0.037	2.587	0.647	0.034	2.825	0.353
2500	0.149	0.150	0.079	1.888	0.944	0.051	2.932	0.733	0.044	3.405	0.426
3000	0.215	0.216	0.110	1.959	0.979	0.064	3.358	0.839	0.061	3.524	0.440

Table 1. Results for the motivating example applying the proposed code generation algorithm

	1 CPU		2 CPU			4 CPU			8 CPU		
N	original	`while`	time[s]	S	E	time[s]	S	E	time[s]	S	E
1024	0.347	0.384	0.235	1.479	0.740	0.178	1.951	0.488	0.187	1.853	0.232
1280	0.658	0.702	0.392	1.676	0.838	0.317	2.075	0.519	0.397	1.657	0.207
1536	1.066	1.177	0.661	1.614	0.807	0.559	1.906	0.477	0.691	1.543	0.193
1792	1.902	2.132	1.480	1.285	0.643	1.281	1.485	0.371	1.206	1.577	0.197
2048	3.210	3.607	2.097	1.531	0.765	1.951	1.646	0.411	1.807	1.777	0.222

Table 2. Results for `LU_HP_pintgr.f2p_2` applying the proposed code generation algorithm

Data in Table 1 demonstrate that there is no significant difference in the execution time of the original `for` loop and that of the corresponding `while`-based loop on a single processor. Increasing the value of *N* results in increasing the speedup and efficiency of the generated parallel code. The reason is the performance of the shared memory parallel program depends considerably on the volume of calculations executed per slice

	1 CPU		2 CPU			4 CPU			8 CPU		
N	original	codegen	time[s]	S	E	time[s]	S	E	time[s]	S	E
1024	0.347	0.359	0.196	1.768	0.884	0.152	2.277	0.569	0.152	2.279	0.285
1280	0.658	0.660	0.389	1.693	0.846	0.301	2.183	0.546	0.394	1.668	0.209
1536	1.066	1.153	0.649	1.643	0.822	0.531	2.009	0.502	0.659	1.618	0.202
1792	1.902	2.031	1.179	1.613	0.807	0.912	2.085	0.521	1.284	1.481	0.185
2048	3.210	3.442	2.025	1.585	0.793	1.594	2.013	0.503	1.798	1.785	0.223

Table 3. Results obtained for `LU_HP_pintgr.f2p_2` using Omega's codegen

(the product of the volume of calculations represented by the loop statements and the number of the loop iterations). We get positive speedup ($S > 1$) when the time of useful calculations (presented by the loop statement instances) is greater than the time overhead incurred by the `while`-based code plus additional thread management and memory bandwidth limitations of the multiprocessor environment.

For all examined NAS benchmarks (8 loop nests in total), we compared the execution times of original loops and those of the `while`-based code on a single processor. These execution times were all very similar (the maximal difference was less than **17%**). Furthermore, the parallelization of all programs resulted in positive speedups.

We report the detailed results for the `LU_HP_pintgr.f2p_2` loop nest of the NAS benchmark. Because this loop features uniform dependences, we were able to generate parallel code not only applying the proposed algorithm but also using Omega's codegen [19, 4]. The results of experiments with these codes are presented in Table 2 and Table 3, respectively. As we can see from these tables, the proposed algorithm does not introduce significant overhead in comparison to Omega's codegen.

We may conclude that while well-known parallelization and code generation techniques fail to extract synchronization free slices described with non-affine forms and having an arbitrary graph topology, the approach presented in this paper permits for producing code scanning such slices. The generated code is based on a `while` loop and its performance is similar to that of code based on a `for` loop.

6 Conclusions and Future Work

In this paper, we have presented an approach to extract synchronization free slices being represented by an arbitrary topology graph of dependent loop statement instances. It permits for generating `while` loop based code that allows us to scan slices even when they are described with non-affine expressions. The proposed algorithm is based on building a single dependence relation describing all the dependences of a slice in the lexicographic order, then this relation is used to produce a `while` loop scanning synchronization free slices of an arbitrary graph topology. We carried out experiments with code generated by means of the presented approach, demonstrating that its performance is similar to that of static control code with `for` loop.

In order to extract sources of synchronization-free slices that are described with non-affine forms and have an arbitrary graph topology, we should be able to calculate

the transitive closure of a union of non-uniform dependence relations. We currently investigate the two following approaches to this problem.

1. The symbolic computation of the exact transitive closure of a parameterized affine relation (such a relation describes an infinite graph) is not possible in general [20]. The case of convex uniform relations is well known and yields affine transitive closures [20], but other relevant cases may allow the symbolic computation of non-affine representations as well. A symbolic representation of non-affine slices may allow for even more efficient code generation schemes.
2. When the exact transitive closure cannot be computed symbolically, one may still try to approximate it. Only very coarse approximations have been proposed so far, especially in the (common) case of non-convex iteration sets. Approximations will lead to suboptimal extraction of parallelism; the induced loss of scalability will need to be investigated.

Acknowledgments

This work was partly supported by the SARC FET-27648 and ACOTES IST-34869 european FP6 projects.

References

1. Ancourt C., Irigoin F., Scanning polyhedra with DO loops, In Proc. of the Third ACM/SIGPLAN Symp. on Principles and Practice of Parallel Programming, ACM Press, pp. 39–50, 1991.
2. Bastoul C., Code Generation in the Polyhedral Model Is Easier Than You Think, In Proc. of the PACT'13 IEEE Intl. Conf. on Parallel Architecture and Compilation Techniques, Juan-les-Pins, pp. 7–16, 2004.
3. Beletska A., Barthou D., Bielecki W., Cohen A., Computing the Transitive Closure of a Union of Affine Tuple Relations. In Proc. of the Third Annual Conference on Combinatorial Optimization and Applications (COCOA'2009), pp. 98–109, LNCS 5573, 2009.
4. Beletska A., Bielecki W., Siedlecki K., San Pietro P., Finding Synchronization-Free Slices of Operations in Arbitrarily Nested Loops. In Proc. of ICCSA, pp. 871–886, LNCS 5073, 2008.
5. Beletska A., Bielecki W., San Pietro P., Extracting Coarse-Grained Parallelism in Program Loops with the Slicing Framework. In IEEE Proc. of ISPDC, page 29, 2007.
6. Beletska A., San Pietro P., Extracting Coarse-Grained Parallelism with the Affine Transformation Framework and its Limitations. Electronic Modelling, no.5, 2006.
7. Bielecki W., Beletska A., San Pietro P., Finding Synchronization-Free Parallelism Represented with Trees of Dependent operations. In Proc. of ICA3PP, pp. 185-195, LNCS 5022, 2008.
8. Bielecki W., Klimek T., Trifunovic K., Calculating Exact Transitive Closure for a Normalized Affine Integer Tuple Relation. Journal of Electronic Notes in Discrete Mathematics 33, pp. 7–14. 2009.
9. Boigelot B., P. Wolper, Symbolic Verification With Periodic Sets. Proc. of the 6th conference on Computer-Aided Verification, pp. 55–76. LNCS 818, 1994.
10. Boulet P., Darte A., Silber G.A., Vivien F., Loop parallelization algorithms: from parallelism extraction to code generation, Parallel Computing 24, pp. 421–444, 1998.

11. Comon H., Jurski Y., Multiple counters automata, safety analysis and Presburger arithmetic. Proc. Computer Aided Verification, pp. 268–279, LNCS 1427, 1998.
12. van Engelen R. A., Efficient symbolic analysis for optimizing compilers. Proc of the Intl. Conf. on Compiler Construction (ETAPS CC'01), pp. 118–132, 2001.
13. Feautrier P., Some efficient solutions to the affine scheduling problem, Part I, one dimensional time, Intl. Journal of Parallel Programming 21, pp. 313–348, 1992.
14. Feautrier P., Some efficient solutions to the affine scheduling problem, Part II, multidimensional time, Intl. Journal of Parallel Programming 21, pp. 389–420, 1992.
15. Feautrier P., Boulet P., Scanning polyhedra without DO-loops, In Parallel Architectures and Compilation Techniques (PACT'98), 1998.
16. Henzinger T. A., A theory of hybrid automata. Symp. on Logic in Computer Science (LICS'96), 1996.
17. Kelly W., Maslov V., Pugh W., Rosser E., Shpeisman T., Wonnacott D.: The Omega library interface guide, Technical Report CS-TR-3445, University of Maryland, 1995.
18. Kelly W., Optimization within a Unified Transformation Framework, Technical Report CS-TR-3725, University of Maryland, 1996.
19. Kelly W., Pugh W., Rosser E., Shpeisman T., Transitive closure of infinite graphs and its applications, Intl. J. of Parallel Programming, 24 (6), pp. 579–598, 1996.
20. Kelly W., Pugh W., Rosser E., Code generation for multiple mappings. Frontiers'95 Symposium on the frontiers of massively parallel computation, 1995.
21. Lee C.G., Stoodley M. , UTDSP Benchmark Suite, Univ. of Toronto, Canada, 1992, `http://www.eecg.toronto.edu/corinna/DSP/infrastructureUTDSP.html`
22. Lim A.W., Lam M., Maximizing Parallelism and Minimizing Synchronization with Affine Transforms. In Proc. of the Symp. on the Principles of Programming Languages, pp. 201–214, 1997.
23. NAS benchmarks suite, `http://www.nas.nasa.gov`
24. The Omega Project, `http://www.cs.umd.edu/projects/omega`
25. Piplib - A parametric integer linear programming solver, `http://www.prism.uvsq.fr/~cedb/bastools/piplib.html`
26. Pean D.-L., Chua H.-T., Chen C., A Release Combined Scheduling Scheme for Non-Uniform Dependence Loops. J. Inf. Sci. Eng. (JISE) 18(2), pp. 223-255, 2002.
27. Prakash S.R., Srikant Y.N., Hyperplane Partitioning: An Approach to Global Data Partitioning for Distributed Memory Machines. IPPS/SPDP 1999:744.
28. Pugh W., Wonnacott D., An exact method for analysis of value-based array data dependences: Workshop on Languages and Compilers for Parallel Computing, 1993.
29. Pugh W., Rosser E., Iteration Space Slicing and Its Application to Communication Optimization, in Proc. of the Intl. Conf. on Supercomputing, pp. 221–228, 1997.
30. Quillere F., Rajopadhye S., Wilde D., Generation of efficient nested loops from polyhedra, Intl. Journal of Parallel Programming 28, 2000.
31. Shen Z, Li Z, Yew P.-C., An Empirical Study of Fortran Programs for Parallelizing Compilers, IEEE Trans. Parallel Distributed Syst., vol. 1, pp. 356–364, July 1990.
32. Vasilache N., Bastoul C., Cohen A., Polyhedral code generation in the real world, In Proc. of the Intl. Conf. on Compiler Construction (ETAPS CC'06), LNCS, pp. 185–201, 2006.
33. Wolfe M. J., Beyond induction variables. In Symp. on Programming Languages and Implementation (PLDI'92), pp. 162–174, 1992.
34. Yu Y., D'Hollander E.H., Non-Uniform Dependences Partitioned by Recurrence Chains, In Proc. the 2004 International Conference on Parallel Processing (ICPP'04), pp.100–107, 2004.