УДК 519.872

W. Bielecki, K. Siedlecki
Faculty of Computer Science, Technical University of Szczecin
(Żołnierska 49 st., 71-210 Szczecin, Poland,
fax. (+4891) 4876439, E-mail: wbielecki@wi.ps.pl, ksiedlecki@wi.ps.pl)

Extracting Synchronization-free Slices in Perfectly Nested Loops

An algorithm, permitting us to extract iterations belonging to synchronization-free slices and to generate code enumerating sources of such slices and iterations of each slice in lexicographical order is presented. Synchronization-free slices can be executed independently preserving the lexicographic order of iterations in each slice. Our approach requires exact dependence analysis and based on operations on relations and sets. To describe and implement the algorithms, the dependence analysis by Pugh and Wonnacott was chosen where dependences are found in the form of tuple relations. The proposed algorithms have been implemented and verified by means of the Omega project software. Presburger arithmetic limitations are discussed. Results of experiments are presented. Tasks for future research are outlined.

Представлен алгоритм, позволяющий выделить итерации, принадлежащие несинхронизированным фрагментам, и генерировать программу, перечисляющую источники таких фрагментов и итераций в каждом фрагменте в лексикографическом порядке. Несинхронизированные фрагменты могут выполняться независимо, сохраняя лексикографический порядок итераций в каждом фрагменте. Данный подход требует точного анализа зависимости и основан на операциях с отношениями и множествами. Для описания и реализации алгоритмов, выбран анализ зависимости по Пугу и Воннакоту, в котором найдены зависимости в форме отношений кортежа. Предложенные алгоритмы реализованы и верифицированы посредством программного пакета Омега. Представлены результаты экспериментов.

K e y w o r d s: loop transformations, perfectly nested loops, synchronization-free parallelism.

Introduction. Extracting synchronization-free slices in loops is of great importance for parallel and distributed computing, enhancing code locality, and reducing memory requirements. Different techniques have been developed to extract synchronization-free parallelism available in loops, for example, [1—13]. However, to our knowledge, none of well-known techniques extracts the entire synchronization-free parallelism available in the general case of affine non-uniform loops.

The goal of this paper is to present an approach which permits us to extract synchronization-free slices available in loops when well-known techniques may fail to extract such slices. It is applicable to perfectly nested both non-parame-

terized and parameterized loops and allows synchronization-free slices to be extracted at compile or run-time.

The purpose of extracting synchronization-free slices is not only to get scalable performance on parallel computers and distributed systems but also to enhance performance on a uniprocessor thanks to enhancing data locality and to reduce memory requirements to decrease cost and power consumption in embedded systems. When independent threads are extracted, well-known approaches can be applied to enhance data locality, for example, blocking and array contraction. Each new value produced at each iteration must be kept into a memory buffer until its last use by another statement of the loop. The size of memory needed to store a value computed and used at different iterations is given by the amount of iterations that the value has to cross. In this paper we describe a technique that permits us to extract synchronization-free slices, each of them is comprised of a chain of dependent iterations, hence each value produced at iteration i is consumed at iteration $i + 1$. This reduces the size of required memory to hold temporary values.

Computations described with a loop can be represented with independent iterations, and (or) synchronization-free treads, and (or) thread requiring synchronization. Our paper deals only with extracting synchronization-free slices. The separation of synchronization-free threads from the rest computation represented with a loop is motivated by the following arguments. Synchronization-free slices can be executed independently from the rest computation represented with a loop. This may permit us to enhance performance on parallel computers and distributed systems (no synchronization at executing on parallel computers, no communications at executing on distributed systems) because computation may be represented only or mostly with synchronization-free slices. Code representing synchronization-free slices can be used to synthesize hardware of an embedded system permitting for reducing the memory size and power consumption, whereas code representing the rest computation can be executed by means of software of the embedded system. To parallelize the rest iterations(independent, i. e., those that do not belong to any slice as well as iterations belonging to slices requiring synchronization), well-known transformations can be applied, for example [11, 12, 14—17] and this task is out of the scope of this paper.

The main idea of the proposed approach is based on the iteration space slicing [18]. It is to firstly split all the loop iterations into dependent and independent ones. Next synchronization-free slices are extracted from dependent iterations. They can be executed independently without any synchronization among them.

Our approach is based on exact data dependence analysis and on operations on sets and relations. We have implemented and verified our approach by means of the Omega project software [19].

Background. In this paper, we deal with affine perfectly nested loop where, for given loop indices, lower and upper bounds as well as array subscripts and conditionals are affine functions of surrounding loop indices and possibly of structure parameters, and the loop steps are known positive constants.

Two iterations I and J are dependent if both access the same memory location and if at least one access is a write. We refer to I and J as the source and destination of a dependence, respectively, provided that I is lexicographically less than $J (I \prec J)$. The vector $d = J - I$ is referred to the dependence vector. The loop nest is said to be uniform if all dependence vectors do not depend neither on I, nor on J.

Our approach requires an exact representation of loop-carried dependences and consequently an exact dependence analysis which detects a dependence if and only if it exists. To describe and implement our algorithms, we chose the dependence analysis proposed by Pugh and Wonnacott [20].

Program slicing is a viable method to restrict the focus of a task to specific sub-components of a program. Program slicing was first introduced by Mark Weiser [21]. According to the original definition [22], the notion of slice was based on the deletion of statements. A slice is an executable subset of program statements that preserves the original behavior of the program with respect to a subset of variables of interest and at a given program point [22].

In paper [18] the idea of iteration space slicing was introduced. Iteration space slicing takes dependence information as input to find all statement instances from a given loop nest which must be executed to produce the correct values for the specified array elements. We can think of the slice as following chains of transitive dependences to reach all statement instances which can affect the result.

In this paper, we deal with specific slices, which can be synchronization-free or requiring synchronization. In our paper [23], we introduced the following definitions.

Definition 1. A slice is a set of dependent iterations including an ultimate dependence source and all the dependence destinations such that each dependence destination except from the lexicographically maximal destination (ultimate dependence destination) is the source of the next dependence.

Definition 2. A slice is independent or synchronization-free if the intersection of the set of iterations representing this slice and the set representing the rest of computation in a loop is empty.

Definition 3. The source of a slice is the ultimate dependence source that this slice comprises, i.e., the lexicographically minimal iteration among all the iterations belonging to this slice.

Our algorithms are based on the operations on relations and sets that are described in detail in [19] and we assume that the reader is familiar with these oper-

ations. We note only that we use both positive transitive closure, R^+, and transitive closure, R^*, in the algorithm presented in this paper.

The next section introduces an algorithm to find iterations belonging to each synchronization-free slice as well as to generate code scanning sources of synchronization-free slices and iterations of each slice in lexicographic order provided that are given sources of synchronization-free slices extracted according to algorithms presented in our paper [23].

Finding iterations of synchronization-free slices and code generation. In this section, we present how to find iterations belonging to synchronization-free slices and how to generate code that scans synchronization-free slices and iterations within each synchronization-free slice in lexicographic order. We use $\{X, Y\}$ to represent the cross-product of sets X, Y, i. e., $\{X, Y\} := X \times Y := \{[x, y] \mid x \in X, y \in Y\}$.

To demonstrate a trouble at generating code scanning synchronization-free slices, let us consider the following dependence relation

$$R1 := \{[i,\ j] \text{ -> } [i + 1,\ \ j + 1] : 1 \le i < 4 \ \&\& \ 1 \le j < 4\}.$$

Set comprising sources of synchronization-free slices, *SFS*, being calculated on the basis of this relation is as follows

$SFS = (\text{domain}\, R1) - (\text{range}\, R1) = \{[1, j]: 1 \le j \le 3\}$ union $\{[i,1]: 2 \le i \le 3\}$. There is no problem to scan iterations of a single slice with applying a code generator to a set

$$R1\ (\{[\text{an element} \in SFS]\}).$$

But if we wish to generate a loop whose outer nests scan sources of slices while inner nests scan iterations of each slices, we have a trouble. The set representing iterations of all slices is as follows

$$S = R1^*(SFS) = \{[i, j]: 2 \le i \le j \le 4\} \text{ union } \{[i, j]: 2 \le j < i \le 4\}$$

$$\text{union } \{[1, j]: 1 \le j \le 3\} \text{ union } \{[i,1]: 2 \le i \le 3\}.$$

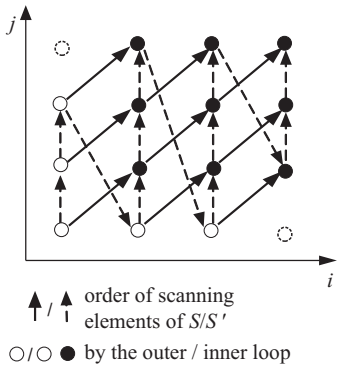
Code, enumerating elements of set S in lexicographic order, scans iterations belonging to different slices and it does not realize our goal (Fig. 1).

To generate correct code, we may apply the following scheme:

Code enumerating iteration vectors for elements of *SFS*;

Code scanning elements of the set $R1^*$ ({[iteration vector of an element of *SFS*]}).

To implement the scheme above, we propose a solution that permits for applying any well-known code generation technique. Our proposal is to form set S' that is the union of all the cross-products $\{[\ e_i, R^*(e_i)]\}$, $i = 1, 2, ..., n$, where n is the number of synchronization-free slices, e_i represents a slice source (the



Code enumerating iterations of set S:

```
for(j = 1; j <= 3; j++)
   s1(1, j);
for(i = 2; i <= 4; i++) {
  if (i <= 3)
          s1(i,1);
  for(j = 2; j <= i–1; j++)
          s1(i, j);
  for(j = i; j <= 4; j++)
          s1(i, j);
}
```

Code enumerating iterations of set S':

```
par for(j = 1; j <= 3; j++)
{
  for(i′ = 1; i′ <= –j + 5; i′++)
          s1(i′, i′+j–1);
}
par for(i = 2; i <= 3; i++)
{
  for(i′ = i; i′ <= 4; i′++)
          s1(i′, i′–i+1);
}
```

Fig. 1. Scanning elements of sets S and S'

i-th-element of set SFS), $R^*(e_i)$ is the set including the iterations belonging to the slice starting with e_i. For relation $R1$, we get the following set

$$S' := \{[1, j, i', i'+j-1]: 2 \le i' \le -j + 5 \;\&\&\; 1 \le j\} \text{ union } \{[i,1, i', i'-i+1]: 2 \le i < i' \le 4\}$$

$$\text{union } \{[1,\ j,\ 1,\ j]: 1 \le j \le 3\} \text{ union } \{[i,\ 1,\ i,\ 1]: 2 \le i \le 3\}.$$

The code, scanning elements of set S', is presented in Fig. 1, where the indices representing slice sources are removed from the loop statement.

Below we attach an algorithm implementing the proposed solution and taking as input the output of Algorithm 1 or Algorithm 2 [23].

Algorithm 1. ***Finding iterations belonging to synchronization-free slices and code generation.***

Input: output of Algorithm 1 [23] (a set of sources of synchronization-free slices, *SFS*, and set S_In containing dependence relations), or output of Algorithm 2 [23] (loop indices defining subspaces where the algorithm extracts sources of synchronization-free slices; sources of synchronization-free slices described with relations contained in set S_Out, *SFS*; set S_Out); the number of dependence relations contained in set S_In or set S_Out, m.

Output: code scanning slices and iterations of each of them in lexicographic order.

1. If the input of the algorithm is the output of Algorithm 2 [23], then

1.1. For each dependence relation in set S_Out remove constraints imposed on the loop indices being produced with Algorithm 2 [23] and declare these indices as symbolic constants.

1.2. Generate a nest of sequential loops scanning the values of the loop indices being produced with Algorithm 2 [23] by means of extracting and rewriting the correspondent nest of the original loop.

2. Calculate relation R as the union of all the relations from set S_In (if the input is the output of Algorithm 1 [23] or set S_Out (if the input is the output of Algorithm 2 [23]) for which $R_i(SFS) \neq$ EMPTY, where R_i is a relation from set S_In or set S_Out, $i \in [1, m]$.

3. Calculate set S using step 3a or 3b.

3a. With applying transitive closure:

3.1. Calculate transitive closure of R, $R^* := R+ \cup I$.

3.2. Build set S of the form $S := \{[e_i , R^*(e_i)]: \forall e_i \in SFS \}$

3b. Without applying transitive closure (only for non-parameterized *SFS* and *R*)

3.1. Form set SFSD as follows:

$SFSD := \{ [e_i, e_i] : \forall e_i \in SFS \}$,

3.2. Form relation *RD* as below:

$RD := \{[e_i, \text{In}]\text{->}[e_i , R (\text{In})]: \forall e_i \in SFS \}$,

where In is an arbitrary tuple of variables of the same size as that of the tuple representing e_i ,

3.3. $S := SFSD$, Temp := *SFSD*,

3.4. Temp := *RD* (Temp) – S,

3.5. If Temp ≠ EMPTY then $S := S \cup$ Temp go to 3.4, else the end.

4. Generate code enumerating elements of set S in lexicographic order by means of any known technique, for example, presented in [3, 24—26].

5. Remove from the loop statements of the code, generated in step 4, the first $n/2$ variables (variables of the tuple representing slice sources).

6. If step 1.2 generates a nest of loops, then inside the nest generated at step 1.2 insert loops scanning synchronization-free slices and iterations of each of them being generated in steps 3—5.

It is worth to note that the first $n/2$ variables of set S built in step 3a or 3b, where n is the number of all the variables of S, represent sources of synchronization-free slices, while the rest $n/2$ variables of this set are responsible for scanning the slice iterations. The code generated from set S (the result of step 4) scans iterations belonging to slices in lexicographic order starting with the lexicographically minimal slice source, but the loop statements in such a code have redundant index variables. The code, generated from set S and where the first $n/2$ variables (responsible for representing slice sources) are removed from the loop statements (the result of step 5) does not have redundant index variables in the loop statements and scans slices and iterations of each of them in lexicographic order.

Step 3b can be applied to loops with parameterized upper bounds only at run-time when the loop bounds become known.

Step 3a of Algorithm 3 is based on applying the transitive closure operation. As it was already mentioned in [23], exact transitive closure cannot be represented with affine forms in the general case of non-uniform dependence relations. When for a given relation, exact transitive closure cannot be calculated by the Omega calculator, there exists a possibility to calculate bounds of transitive closure[27] and still find the synchronization-free slices.

Let us illustrate Algorithm 3 by means of the following example.

Example 1.

for(i = 1; $i \le n$; i++)
  for(j = 1; $j \le n$; j++)
   $a(j+4, j+1) = a\ (i+2*j+1, i + j + 3)$

For the loop of Example 1, Petit generates the following dependence relations

$R1 := \{[i,5] \text{ -> } [i, i + 7] : 1 \le i \le n - 7\}$,
$R2 := \{[i,5] \text{ -> } [i', i + 7] : 1 \le i < i' \le n \ \&\& \ i \le n - 7\}$,
$R3 := \{[i,i + 8] \text{ -> } [i + 1,5] : 1 \le i \le n - 8\}$,
$R4 := \{[i,j] \text{ -> } [j - 7,5] : 1 \le i \le j - 8 \ \&\& \ j \le n\}$,
$R5 := \{[i,j] \text{ -> } [i', j] : 1 \le i < i' \le n \ \&\& \ 1 \le j \le n\}$.

After removing redundant transitive dependences represented with $R5$, we get $R5'$ in the form

$$R5' := \{[i,j] \text{ -> } [i+1,j] : 1 \le i < n \ \&\& \ 1 \le j \le n\}.$$

Fig. 2 represent dependences for Example 1 before and after removing redundant dependences (in Fig. 2, *a*, dependences described with $R5$ are shown only for $j = 1$).

Following Algorithm 1 [23], we yield.

1. $R := R1$ union $R2$ union $R3$ union $R4$ union $R5' = \{[i,5] \text{ -> } [i,i+7] : 1 \le i \le \le n-7\}$ union $\{[i,5] \text{ -> } [i',i+7] : 1 \le i < i' \le n \ \&\& \ i \le n - 7\}$ union $\{[i,j] \text{ -> } [j-7,5] : 1 \le i \le j-8 \ \&\& \ j \le n\}$ union $\{[i,j] \text{ -> } [i+1,j] : 1 \le i < n \ \&\& \ 1 \le j \le n\}$,

$IR :=$ inverse $R := \{[i,i+7] \text{ -> } [i,5] : 1 \le i \le n-7\}$ union $\{[i,j] \text{ -> } [j-7,5] : 8 \le j \le \le i+6, n \ \&\& \ i \le n\}$ union $\{[i,5] \text{ -> } [i',i+7] : 1 \le i' < i \le n-7\}$ union $\{[i,j] \text{ -> } [i-1,j] : 2 \le i \le n \ \&\& \ 1 \le j \le n\}$,

$CI :=$ EMPTY.

2. $CI := \{[i, i+7]: 1 \le i \le n-7\}$ union $\{[i,5]: 1 \le i \le n-7\}$ union $\{[i,j]: 8 \le j \le \le i+6, n \ \&\& \ i \le n\}$ union $\{[i,j]: 1 \le i \le j-8 \ \&\& \ j \le n\}$.

3. $UDS :=$ domain R – range $R := \{[1,j]: 1 \le j \le 7, n \ \&\& \ 2 \le n\}$ union $\{[1,j]: 9 \le \le j \le n\}$.

4. Since CI is not EMPTY, go to step 5a (using transitive closure).

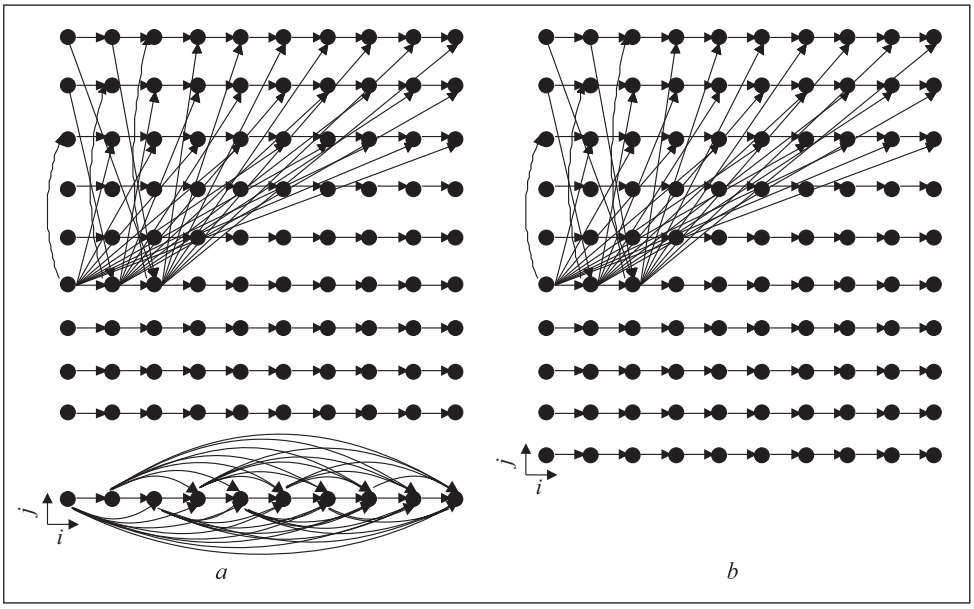


Fig. 2. Dependences described by relations $R1$— $R5$ (*a*) and $R1$—$R4$ and $R5'$ (*b*) for $n = 10$

5. The omega calculator does not permit us to calculate exact transitive closure for *IR*. Calculating and using the upper bound (UNKNOWN=True) for the inexact transitive closure yielded with the Omega calculator, we get the following set

$$SFS := \{[1, j]: j, 2 \le n \le 7 \;\&\&\; 1 \le j\} \text{ union } \{[1, j]: n = 8 \;\&\&\; 6 \le j \le 7\}$$
$$\text{union } \{[1, j]: n = 8 \&\& 1 \le j \le 4\},$$

which does not include all sources of synchronization-free slices, we miss some sources.

Calculating the lower bound for *TIR* (UNKNOWN=False) permits us to extract all sources of synchronization-free slices being contained in the following set

$$SFS \{[1, j]: j, 2 \le n \le 7 \;\&\&\; 1 \le j\} \text{ union } \{[1, j]: 6 \le j \le 7 \;\&\&\; 8 \le n\}$$
$$\text{union } \{[1, j]: 1 \le j \le 4 \;\&\&\; 8 \le n\}.$$

Applying Algorithm 1 with step 3a, we get the following code scanning slices and iterations of each of them (Fig. 3)

Applying Algorithm 1 to the output of Algorithm 2 [20] for Example 1 being presented in [23], we generate the following code

```
if (n >= 2)
 seqfor(k = 1; k <= n; k++)
  parfor(i = 1; i <= n; i++)
```

```
if (n i 2 && n ≤ 7) {
    par  for(t2  =  1;  t2  ≤  n;
          t2++) {
    s1(1,t2);
    seq for(t3 =2; t3 ≤ n;
          t3++)
          s1(t3,t2);
  }
}
```

```
if (n >= 8) {
    par for(t2 = 1; t2 <= 4;
          t2++) {
    s1(1,t2);
    seq for(t3 = 2; t3 ≤ n;
          t3++)
    s1(t3,t2);
  }
}
```

```
if (n >= 8) {
    par for(t2 = 6; t2 ≤ 7;
          t2++) {
    s1(1,t2);
    seq for(t3 =2; t3 ≤ n;
          t3++)
    s1(t3,t2);
  }
}
```

Fig. 3. Cod scanning slices and iterations

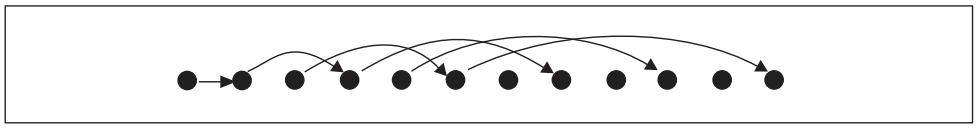


Fig. 4. Dependences for the loop of Example 2 for n=12

```
seqfor(j = 1; j <= n; j++)
  a(i,j,k)=a(i,j–1,k)
  b(i,j,k)=b(i,j,k–1).
```

This loop enumerates n synchronization-free slices defined by values of indices i and j in each of n subspaces being defined by values of k. Subspaces are scanned sequentially with the most outer loop.

Presburger arithmetic limitations. In paper [23], several limitations of Presburger arithmetic concerned calculating exact transitive closure were mentioned. In this section, we show why using only Presburger arithmetic does not permit us to extract the entire synchronization-free parallelism available in an examined loop. Then we discuss what problems have to be resolved in the future to apply the presented algorithms for extracting slices in loops whose exact transitive closures are represented with non-linear forms.

Example 2. Let us consider the following loop:

```
for(i = 1; i ≤ n; i++)
  a[i] = a[2i];
```

The relation representing loop-carried dependences in this loop is as follows

$$R := \{[i]\text{->}[2i]\colon 1 \le i,\ 2i \le n\}.$$

Fig. 4 shows the dependences among iterations for the loop of Example 2. The sources of synchronization-free slices belong to the following set, SFS,

$SFS := UDS :=$ domain R – range $R :=$ {[i] : Exists (alpha : 2alpha = 1+ +i &&1≤ i && 2i ≤ n)}.

The transitive closure of relation R, calculated with the Omega calculator, is of the form

$TR := (R+)$ union $\{[i]\text{->}[i]\} = \{[i] \text{->} [i]\}$ union $\{[i] \text{->} [i'] : \text{Exists (alpha} : 0 = i' + \text{+ 2alpha \&\& } 4i \le i' \le n \text{ \&\& } 1 \le i \text{ \&\& } 2i' \le n + 8i \text{ \&\& UNKNOWN)}\}$ union $\{[i] \text{->} \text{->}[2i] : 1 \le i \text{ \&\& } 2i \le n\}$ union $\{[i] \text{->} [4i] : 1 \le i \text{ \&\& } 4i \le n\}$.

The upper bound (UNKNOW = TRUE) and the lower bound (UNKNOWN = FALSE) of *TR*, *UTR* and *LTR*, are as follows

$UTR :=$ upper_bound $TR := \{[i] \text{->} [i]\}$ union $\{[i] \text{->} [i'] : \text{Exists (alpha} : 0 = i' + \text{+2alpha \&\& } 4i \le i' \le n \text{ \&\& } 1 \le i \text{ \&\& } 2i' \le n + 8i)\}$ union $\{[i] \text{->} [2i] : 1 \le i \text{ \&\& } 2i \le n\}$,

$LTR :=$ lower_bound $TR := \{[i] \text{->}[i]\}$ union $\{[i] \text{->} [2i] : 1 \le i \text{ \&\& } 2i \le n\}$ union $\{[i] \text{->} [4i] : 1 \le i \text{ \&\& } 4i \le n\}$.

The code generated with applying the upper bound of *TR* is incorrect while that generated with applying the lower bound of *TR* does not enumerate all iterations of the slice with source at iteration 1.

To generate correct code, we have calculated the exact positive transitive closure manually, it is of the form

$$R+:= \{[i] \text{->} [j] : \text{Exists} (k: k \ge 1 \text{ \&\& } j = 2^k * i \text{ \&\& } 1 \le i, j \le n)\}.$$

The transitive closure is represented with the following set

$$R^* := R+ \text{ union } I := \{[i] \text{->} [j] : \text{Exists} (k: k \ge 0 \text{ \&\& } j = 2^k * i \text{ \&\& } 1 \le i, j \le n)\},$$

where I is the identity relation. Following Algorithm 3 and using set SFS and relation R^*, we build the following set, *S*, representing synchronization-free slices and iterations of each of them

$$S := \{[i, j]: \text{Exists (alpha} : 2\text{alpha} = 1 + i \text{ \&\& } 1 \le i \text{ \&\& } 2i \le n) \text{ \&\& exists } (k: k \ge \ge 0 \text{ \&\& } j = 2^k * i \text{ \&\& } 1 \le i, j \le n)\}.$$

As we can see, set *S* (when $n = 12$) represents the three synchronization-free slices: (i): 1,1; 1,2; 1,4; 1,8; (ii):3,3; 3,6; 3,12; (iii): 5,5; 5,10. After applying steps 4 and 5 of Algorithm 1, we yield the following loop scanning synchronization-free slices and their iterations in lexicographic order:

```
par for(t1 = 1; t1 ≤ intDiv(n,2); t1 += 2)
  seq for(t2=0;2^t2 *t1 ≤ n ; t2++) {
     i=2^t2*t1;
     a[i] = a[2i];
  }
```

From the example above, we can observe that the approach presented in this paper works also in the case when the Omega calculator cannot calculate exact transitive closure. To permit us to extract synchronization-free parallelism avail-

able in the affine non-uniform loop by means of the approach presented, we need: (i) calculating exact transitive closure which can be presented not only with affine expressions but also with non-linear ones (Algorithms 1 presented in our paper [23] and Algorithm 1 presented in this paper); (ii) calculating results of the intersection and difference operations on sets which elements are described not only by affine constraints but also with non-linear ones (Algorithm 1 [23]); (iii) code generation for sets of iterations which elements are described not only by affine constraints but also with non-linear ones (Algorithm 3). Resolving these problems is out of the scope of this paper. We plan to study them in our future work.

Experiments. The approach, described in this paper, was implemented using the Omega project software[19]. We have built a tool that extracts synchronization-free slices from perfectly nested loops represented in the Petit language[19].

To evaluate the effectiveness of the proposed approach, we have carried out experiments with the perfectly nested Livermore loops [28] using the tool developed by us. Table 1 comprises the results of our experiments. The first column presents the names of Livermore loops. All the loops under our experiments have parameterized upper bounds and non-parameterized lower bounds. To reduce the number of dependences or to permit us to find dependences with Petit, we have transformed several loops manually applying such well-known transformations as scalar expansion (for loop K10 and loop K23), constant propagation (for loop K8), and the substitution of non-linear functions in the loop body by linear ones with the same arguments (for loop K22).

The last transformation was applied to permit us to find dependences with Petit, the body of the loop transformed has the original non-liner function $\exp(x)$. The second column shows whether a source loop was modified or not. The third column indicates whether there exists the necessity in applying step 5a of Algorithm 1 [23] (because the loops under our experiments are parameterized, only step 5a of Algorithm 1 can be applied). In our experiments, only step 3a of Algorithm 3 was applied. The fourth column indicates which algorithm (Algorithm 1 or Algorithm 2 [23]) was applied for extracting sources of synchronization-free slices. The fifth column presents the numbers of synchronization-free slices extracted by our approach in a correspondent loop. The denotation like «$\mathsf{loop} \geq 2 \Rightarrow 1$ in each of loop subspaces» means that if the condition $\mathsf{loop} \geq 2 \Rightarrow 1$ holds, where loop is the upper bound of a correspondent loop, then the number of synchronization-free slices, extracted for this loop in each of loop subspaces, equals 1. The sixth column presents the amount of the time required for extracting synchronization-free slices and generating code by means of our tool on a PC computer with the following characteristics: Athlon 1600 + (~1400 MHz), RAM -256 MB, OS -Windows 2000.

As we can see from Table, the technique presented in this paper extracts slices for each of the loops. In each subspace, slices can be executed independently, while subspaces are scanned sequentially in lexicographic order.

From the sixth column of Table, we may observe that the amounts of the time needed for finding dependence relations and extracting synchronization-free parallelism available in the Livermore loops investigated are not so dramatic. Taking into account that the performance of computers is permanently increased, we have the optimism that the approach presented in this paper may be implemented in both academic and industry compilers in the future.

Related work. The results of the paper are within the iteration space slicing framework introduced by Pugh and Rosser in paper [18]. This framework might have a number of uses. Pugh and Rosser examined in paper [18] how to apply this framework to optimization of interprocessor communication. In particular, they demonstrated how to use slicing to enable loop fusion, tolerate message la-

Results of the experiments

Loop name	Modified	Step 5 of Algorithm 1	Algorithm from [23]	The number of synchronization-free slices	Loop transform time, *s*
K1— hydro fragment	No	No common iterations	1	Loop ≥ 2 ⇒ n	0,07
K5— tri-diagonal elimination, below diagonal	No	5a	2	Loop ≥ 2 ⇒ 1 in each of loop subspaces	0,29
K6 — general linear recurrence equations	No	5a	2	i ≥ 2 ⇒ 1 in each of loop*n subspaces	0,38
K7 — equation of state fragment	No	No common iterations	1	Loop ≥ 2⇒ n	0,06
K8 — ADI integration	Yes	5a	2	n – 1 in each of loop subspaces	0,21
K9 — integrate predictors	No	No common iterations	1	Loop ≥ 2 ⇒ n	0,07
K10 — difference predictors (vec)	Yes	No common iterations	1	Loop ≥ 2 ⇒ n	0,08
K12 — first difference	No	No common iterations	1	Loop ≥ 2 ⇒ n	0,07
K21 — matrix*matrix product	No	5a	2	n ≥ 1 ⇒ 24*n in each of loop subspaces	0,27
K22 — Planckian distribution	Yes	No common iterations	1	Loop ≥ 2⇒ n	0,07
K23 — 2-D implicit hydrodynamics fragment	Yes	5a	2	n ≥ 3 ⇒ 1 in each of 5* loop subspaces	0,46

tency and allow message coalescing. To form slices, they use the transitive closure operation to compute the transitive dependences among iterations and then compute the set of iterations that are reachable via the transitive closure in the forwards or backwards direction, depending on the application. But Pugh and Rosser do not describe in paper [18] how to construct synchronization-free slices. In our paper [23], we presented how to extract sources of both synchronization-free slices and slices requiring synchronization.

As far as concerned code generation, we presented troubles at generating code scanning synchronization-free slices and iterations of each slice. We demonstrate how they can be resolved with applying well-known code generation techniques, for example, [3, 24—26].

Unimodular loop transformations [4, 13], permitting the outermost loop in a nest of loops to be parallelized, find synchronization-free parallelism. But unimodular transformations do not allow such transformations as loop fission, fusion, scaling, reindexing, or reordering to be applied to extract synchronization-free parallelism.

Paper [6] describes an approach, based on Hamiltonian recurrences, permitting us to extract the entire synchronization-free parallelism. But this approach is applicable only to uniform non-parameterized loops.

The affine partitioning framework, considered in many papers, for example, [10—12, 14—16] unifies a large number of previously proposed loop transformations. Today, it is one of the most powerful frameworks for loop transformations allowing us to extract synchronization-free parallelism presented in loops with both uniform and affine dependences. However, for the general case of non-uniform loops, this framework does not permit us to build non-affine schedules for extracting parallelism. We believe that our contribution consists in demonstrating that extracting synchronization-free slices may require non-affine schedules as well as in exposing problems to be resolved in the future to allow for deriving such schedules.

We would like to note that the iteration space slicing is a finer-grained approach than the affine partitioning framework because the presented technique extracts some subset of the statements in a loop body and a subset of the loop nest's iterations while that acts on all statements in a loop, or all iterations of an individual statement. This insight was first presented by Pugh and Rosser in paper [18]. The synchronization-free slice extracted by the technique presented in this paper is not the same as the independent thread extracted by an affine transform[11, 12]. Each slice is a chain of dependent iterations, whereas an independent thread would be comprised of many synchronization-free slices. Each synchronization-free slice is an independent thread but an independent thread would not be a slice. This property makes the iteration space slicing framework more

extendible than the affine partitioning framework and could allow us to extract synchronization-free slices when the affine partitioning framework fails to extract those available in loops. Loops scanning slices are more preferable than loops scanning independent threads when the goal of the loop transformation is to enhance data locality or reduce memory requirements because at executing a slice each value produced at iteration i is consumed at iteration $i + 1$, whereas for an independent thread it is not always true.

The approach, presented in [9], permits for building schedules that can be expressed in Presburger arithmetic only and it does not permit for non-affine schedules.

Conclusion. In paper [23] and in this paper, we described algorithms, permitting us to find synchronization-free slices comprised of iterations of perfectly nested uniform and non-uniform loops. Extracting synchronization-free slices permits us to get code to be executed without synchronization among slices, enhance code locality, and (or) reduce memory requirements. The presented technique comprises the following steps: finding exact dependence relations, removing redundant dependence relations or removing redundant dependences from dependence relations, finding sources of synchronization-free slices, extracting iterations of synchronization-free slices, and code generation. We have implemented the presented technique using the Omega project software and carried out experiments with a number of loops.

The presented algorithms are not only to extract parallelism in loops but also to transform loops to enhance data locality and reduce memory requirements and can be used for uniprocessors as well.

It is worth to note that focusing on synchronization-free parallelism is very important for understanding such a kind of the parallelism available in the loop, not necessarily for its run-time execution. Hence, the results of this paper not necessarily aim at the practical use, but have also a theoretical character. They will permit us to answer the following questions: how many synchronization-free slices are available in the loop; what is the cost we should pay for extracting parallelism and generating code scanning synchronization-free slices available in the loop; whether this code is effective; how many synchronization-free slices are not exposed by means of well-known approaches as well as some other questions.

The most neuralgic operation, used in the presented algorithms, is transitive closure. For the general case of the affine integer tuple relation, exact transitive closure of the relation may not be affine [27]. In such cases, the Omega project software may not permit us to extract synchronization-free parallelism available in non-uniform loops because of the Presburger arithmetic limitations discussed in this paper. There is the need for developing methods and algorithms allowing

us to calculate: (i) exact transitive closure when it cannot be represented only with affine forms; (ii) results of the intersection and difference operations on sets whose elements are described not only by affine constraints but also by non-linear ones. There is the necessity in developing code generation algorithms to be applied to sets of iterations with non-linear constraints. We plan to study these problems in our future research.

Наведено алгоритм, що дозволяє виділити ітерації, які належать несинхронізованим фрагментам, і генерувати програму, яка перелічує джерела таких фрагментів та ітерацій у кожному фрагменті в лексикографічному порядку. Несинхронізовані фрагменти можуть виконуватися незалежно, зберігаючи лексикографічний порядок ітерацій у кожному фрагменті. Даний підхід потребує точного аналізу залежності і базується на операціях з відношенням та множинами. Для описування та реалізації алгоритмів обрано аналіз залежності за Пуго та Воннакотом, у якому знайдено залежності у формі відношень кортежа. Запропоновані алгоритми реалізовано і верифіковано за допомогою програмного пакета Омега. Наведено результати експериментів.

1. *Allen R., Kennedy K.* Optimizing Compilers for Modern Architectures. — San-Francisco: Morgan Kaufmann Publishers Inc., 2001. — 790 p.
2. *Amarasinghe S. P., Lam M. S.* Communication optimization and code generation for distributed memory machines//Proc. of the SIGPLAN'93 New Mexico, June, 1993. — P. 126— 138.
3. *Ancourt C., Irigoin F.* Scanning polyhedra with do loops// Proc. of the Third ACM/SIGPLAN Symposium on Principles and Practice of Parallel Programming. April 21-24. — Virginia : ACM Press, 1991. — P. 39—50.
4. *Banerjee U.* Unimodular transformations of double loops// Proc. of the Third Workshop on Languages and Compilers for Parallel Computing. August, 1990. — Irvine, CA. — P. 192 — 219.
5. *Beletskyy V.* Finding Synchronization-Free Parallelism for Non-uniform Loops// Proc. of the Computational Science — ICCS'2003. Lecture Notes in Computer Science. — Berlin/Heidelberg: Springer, 2003. — V. 2658. — P. 925—934.
6. *Gavaldà R., Ayguadé E., Torres J.* Jbtaning Synchronization-Free Code with Maximum Parallelism//Technical Report LSI-96-23-R. — Barselona: Universitat Politècnica de Catalunya, 1996. — 96 p.
7. *Griebl M., Lengauer C.* Classifying Loops for Space-Time Mapping//In Proc. of the Euro-Par 1996. Lecture Notes in Computer Science. — Berlin : Springer-Verlag, 1996. — P. 467—474.
8. *Huang C., Sadayappan P.* Communication-free hyperplane partitioning of nested loops// J. of Parallel and Distributed Computing. — 1993. — № 19. —P. 90—102.
9. *Kelly W., Pugh W.* Minimizing communication while preserving parallelism//Proc. of the 1996 ACM International Conference on Supercomputing. — Philadelphia, USA, 1996. — P. 52 — 60.
10. *Lim W., Lam M.S.* Communication-free parallelization via affine transformations//Proc. of the Seventh workshop on languages and compilers for parallel computing. — Ithaca, NY, USA, 1994. — P. 92—106 .
11. *Lim W., Cheong G. I., Lam M. S.* An affine partitioning algorithm to maximize parallelism and minimize communication//Proc. of the 13-th ACM SIGARCH. International Conference on Supercomputing. Rhodes, Greece, 1999. — P. 228—237.
12. *Lim W., Lam M. S.* Maximizing parallelism and minimizing synchronization with affine transforms// Conf. Record of the 24-th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Paris, France, 1997. — P. 201—214.

13. *Wolf M. E.* Improving locality and parallelism in nested loops: Ph.D. Dissertation CSL-TR-92-538. — Stanford University. Dept. Computer Science,1992.
14. *Darte A., Robert Y., Vivien F.* Scheduling and Automatic Parallelization.— Boston : Birkhäuser, 2000. — 294 p.
15. *Feautrier P.* Some efficient solutions to the affine scheduling problem, part i, one dimensional time// International J. of Parallel Programming — 1992. — Vol. 21, № 5. — P. 313— 348.
16. *Feautrier P.* Some efficient solutions to the affine scheduling problem, part ii, multidimensional time// Ibid. — P. 389— 420.
17. *Feautrier P.* Toward automatic distribution//J. of Parallel Processing Letters. — 1994. — Vol. 3, № 4. — P. 233—244.
18. *Pugh W. , Rosser E.* Iteration Space Slicing and Its Application to Communication Optimization//Proc. of the International Conference on Supercomputing, July 7—11, Vienna, Austria, 1997. — P. 221—228.
19. *Kelly W., Maslov V., Pugh W. et all.* The omega library interface guide// Technical Report CS-TR-3445. — University of Maryland, College Park, USA, 1995. — 33 p.
20. *Pugh W., Wonnacott D*. Constraint-based array dependence analysis//ACM Trans. on Programming Languages and Systems. — 1998. — Vol. 20, № 3. — P. 635—678.
21. *Weiser M.* Program slices: formal, psychological, and practical investigations of an automatic program abstraction method // PhD thesis. University of Michigan, Ann Arbor, MI, 1979.
22. *Weiser M.* Program Slicing//IEEE Transactions on Software Engineering. — 1984. —V. SE-10, №. 7. — P. 352—357.
23. *Bielecki V., Siedlecki K.* Finding Sources of Synchronization-free Slices in Perfectly Nested Loops // Electronic Modeling. — 2007. — Vol. 29, № 3. — P. 41—53.
24. *Boulet P., Darte A., Silber G. A., Vivien F.* Loop parallelization algorithms: from parallelism extraction to code generation// Parallel Computing, 1998. — № 24. — P. 421—444.
25. *Collard J. F., Feautrier P., Risset T.* Construction of do loops from systems of affine constraints. Construction of Do Loops from Systems of Affine Constraints//Parallel Processing Letters. — 1995. — №5.— P. 421—436.
26. *Quillere F., Rajopadhye S., Wilde D.* Generation of efficient nested loops from polyhedra// International J. of Parallel Programming. — 2000. — Vol. 28, № 5. — P. 469—498.
27. *Kelly W., Pugh W., Rosser E., Shpeisman T.* Transitive Closure of Infinite Graphs and its Applications// Intern. J. of Parallel Programming. — 1996. — V. 24, № 6. — P. 579—598.
28. *Netlib* Repository at UTK and ORNL. http://www.netlib.org/benchmark/livermorec.

Поступила 07.11.06