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

Finding Sources of Synchronization-free Slices in Perfectly Nested Loops

Algorithms, permitting us to find sources of synchronization-free slices of perfectly nested uniform and non-uniform loops, are presented. Sources extracted are to be used for creating synchronization-free-slices that 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.

Представлены алгоритмы, позволяющие находить несинхронизированные фрагменты, содержащие итерации полностью вложенных однородных и неоднородных циклов. Такие фрагменты могут выполняться независимо, сохраняя лексикографический порядок итераций в каждом фрагменте. Предложенный подход основан на операциях отношений и множеств и требует точного анализа зависимостей между операторами программы. Для описания и реализации алгоритмов выбран анализ зависимости по Пугу и Воннакоту, согласно которому зависимости отыскиваются в форме отношений кортежа. Описанные алгоритмы реализованы и верифицированы посредством программного пакета Omega project.

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

Introduction. Finding 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-parameterized 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.

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 [14].

Background. In this paper, we deal with affine loop nests 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. Presented algorithms are to be used for extracting synchronization-free slices. The iterations belonging to slices requiring synchronization can be parallelized with well-known techniques, for example, [1, 15 —19].

For the perfectly nested loop, all its statements are comprised within the innermost nest. We refer to a particular sequential execution of all the statements of the loop body as an iteration. Each iteration of an n-level nested loop is represented with an iteration vector I of dimension n.

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] where dependences are represented with dependence relations comprised of Presburger formulas, which can be built up out of linear constraints over integer variables, logical connectives, and universal and existential quantifiers [20]. We assume that the reader is familiar with that dependence analysis.

We refer to the source (destination) of a dependence as the ultimate dependence source (destination) if it is not the destination (source) of any other dependence. 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 [23] 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.

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 presented in Table 1, where R and S denote relations and sets, respectively. In detail, these operations are described in [14] and we assume that the reader is familiar with these operations. We would like to note only that there exist two relations related to transitive closure: positive transitive closure, $R+$, and transitive closure, $R^* = R+ \cup I$, where I is the identity relation, and both of them are used in the algorithms presented in this paper.

In one of the algorithms presented in this paper, the loop interchange transformation is applied [1]. It consists in switching the nesting order of two loops in a perfect nest. The legality condition of this transformation can be found in [1].

Table 1. **Operations on relations and sets**

Operations	Denotation in the Omega Calculator
$R_1 \cap R_2$ $S_1 \cap S_2$	Intersection
$R_1 - R_2$ $S_1 - S_2$	– , Difference
$R_1 \cup R_2$ $S_1 \cup S_2$	Union
Inverse R	¬ , Inverse
Positive Transitive Closure of R	+, Transitive_Closure

Given dependence relations calculated for the loop, our approach to extracting sources of synchronization-free slices of the loop iterations consists of the following steps. First to increase the number of synchronization-free slices, we should remove redundant dependences. Second, we have to extract a set of all ultimate dependence sources and next split it into two sets including sources of slices requiring synchronization and sources of synchronization-free slices, respectively. When the sources cannot be extracted in the whole loop domain, we try to find subspaces in the loop domain where synchronization-free slices can be extracted.

The following sections describe each step in detail.

Removing redundant dependences. A redundant dependence is one that can be eliminated without missing any information about dependences required for extracting all synchronization-free slices available in the loop. A dependence is redundant if it is implied by the other dependences. For example, a redundant dependence is a direct dependence that is described by different dependence relations (one dependence has to be retained, the rest dependences can be removed) or it is such a direct dependence whose source and destination are the source and destination of a transitive dependence. The elimination of redundant dependences may increase the number of synchronization-free slices that we are able to extract from the loop. Let us consider the following example.

Example 1.

```
    for(i = 1; i ≤ 10; i++)
    for(j = 1; j ≤ 10; j++) {
(a) a[i][j] = a[i] [j –1];
(b) c[i][j] = c[i] [j –1];
(c) b[i][j] = b[i] [j –2];
   }
```

The loop above originates the following dependence relations found with Petit (Fig. 1, *a*)

(data dep. a→a) $R_1 := \{[i,j] \rightarrow [i,j+1] : 1 \le i \le 10 \;\&\&\; 1 \le j \le 9\}$,

(data dep. b→b) $R_2 := \{[i,j] \rightarrow [i,j+1] : 1 \le i \le 10 \;\&\&\; 1 \le j \le 9\}$,

(data dep. c→c) $R_3 := \{[i,j] \rightarrow [i,j+2] : 1 \le i \le 10 \;\&\&\; 1 \le j \le 8\}$.

If we take into account all dependence relations, the presented algorithm cannot extract synchronization-free slices because there exist common dependence destinations described with the different dependence relations. But R_2 is redundant because it represents the same dependences as R_1 does, hence it can be removed (Fig. 1, *b*). Relation R_3 is also redundant because for each direct dependence represented with R_3 there exists a transitive dependence represented with relation R_1, hence R_3 can be removed. Removing R_2 and R_3 permits us to extract synchronization-free slices shown in Fig. 1, *c*.

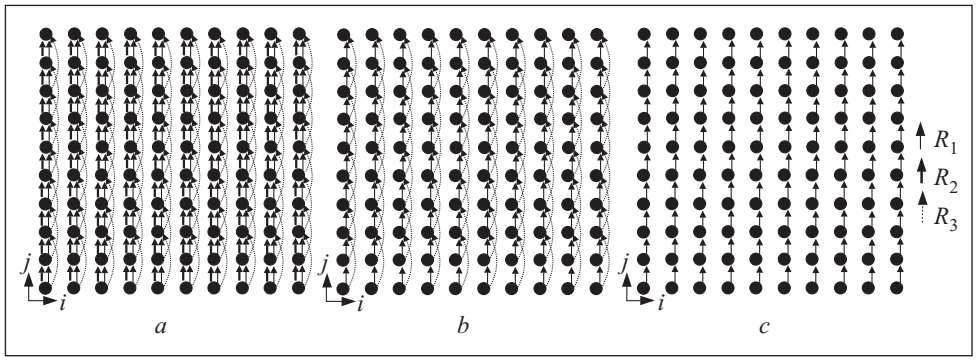

Fig. 1. Dependences for Example 1: *a* — all dependences; *b* — dependences after removing R_2; *c* — dependences after removing R_2 and R_3

The removal of redundant dependences is a well-known problem considered in many publications, for example, in [24] and it is out of the scope of this paper.

Finding sources of slices. In this section, we describe an algorithm that permits us to find the lexicographically minimal iteration among all the iterations belonging to the slice. Such an iteration is the source of a slice.

The following steps are to be fulfilled for extracting synchronization-free slices. First we form two sets containing independent and dependent iterations, respectively. Next using the set, containing dependent iterations, we extract all ultimate dependence sources. Then a set comprising sources for slices requiring synchronization, *RSS*, and a set including sources for synchronization-free slices, *SFS*, are built. The number of iterations in set *SFS* determines the number of synchronization-free slices.

In the algorithm that follows, we suppose that (i) each dependence relation does not represent two or more different dependence destinations corresponding to the same dependence source; if this is the case, the relation has to be normalized to satisfy the above condition (redundant dependences have to be removed or it has to be split into several dependences such that each of them does not describe common iterations); (ii) any two relations, R_i and R_j, such that one of them describes ultimate dependence destinations that are ultimate dependence sources described with the other relation are represented with a single relation $R:=R_i \cup R_j$.

Algorithm 1. ***Extracting sources of slices requiring synchronization and sources of synchronization-free slices***.

Input: set S_In including normalized relations representing loop-carried dependences: $R_1, R_2, \ldots, R_m$, where m is the number of relations.

Output: a set of sources of slices requiring synchronization, *RSS*; a set of sources of synchronization-free slices, *SFS*.

1. Calculate the union of all dependence relations, R, inversion of R, IR, and initialize a set of common iterations, CI:

$$R := R_1 \cup R_2 \cup \ldots \cup R_m ; \; IR := \text{inverse } R ; \; CI := \text{EMPTY}.$$

2. Find a set of all common sources and destinations, CI. If set S_In includes the only relation, then go to step 3. Otherwise for each pair of relations R_i and R_j in set S_In, where $i \neq j$, $i, j \leq m$ do

$$CI = CI \cup (\text{domain } R_i \cap \text{domain } R_j) \cup (\text{range } R_i \cap \text{range } R_j).$$

3. Find dependence sources, I, as the domain of relation R; find dependence destinations, J, as the range of relation R. Find all ultimate dependence sources, UDS, as the difference between sets I and J, that is,

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

4. If CI == EMPTY, then UDS contains all sources of synchronization-free slices, $SFS := UDS$, the end; otherwise go t step 5.

5. Calculate a set of dependence sources belonging to slices requiring synchronization, $RSS1$, using step 5*a* or 5*b* (Table 2).

6. To find a set comprising sources of dependent slices, RSS, calculate the intersection between $RSS1$ and UDS

$$RSS := RSS1 \cap UDS.$$

7. To find a set including sources of synchronization-free slices, SFS, calculate the difference between UDS and RSS

$$SFS := UDS - RSS.$$

It is worth to note that when a set of common iterations, CI, is EMPTY, all ultimate dependence sources are the sources of synchronization-free slices. When the loop originates common iterations, Steps 1 to 5 of Algorithm 1 find all the iterations, $RSS1$, belonging to slices requiring synchronization on the back-

Table 2. **Calculating a set of dependence sources**

Step 5*a*	Step 5*b*
With applying transitive closure	Without applying transitive closure (only for non-parameterized CI and IR)
Calculate: $IR+$:= Transitive Closure IR, $RSS1 := IR+(CI)$ /*applying relation $IR+$ to set CI*/	TEMP := CI, $RSS1$:= CI, L: find a set of dependence sources belonging to slices requiring synchronization, TEMP, as: TEMP := IR(TEMP) /* applying relation IR to set TEMP*/ If TEMP == EMPTY, then go to step 6, otherwise $RSS1 := RSS1$ TEMP, go to L:

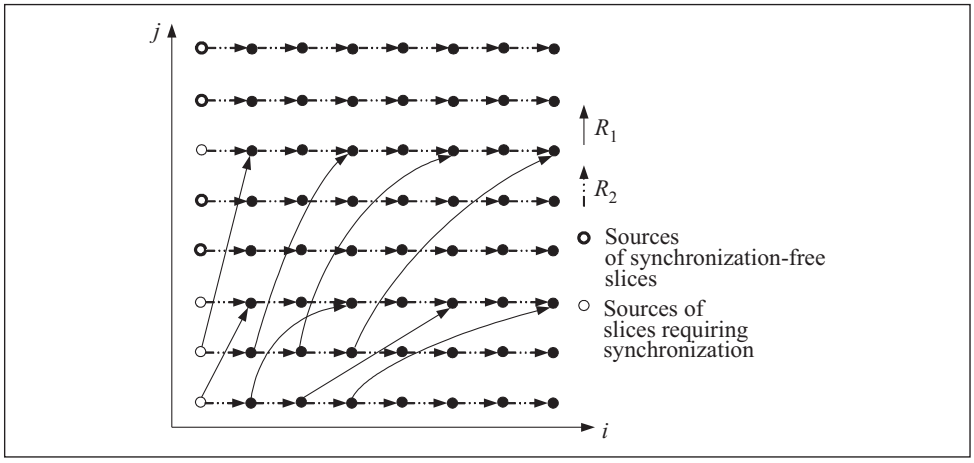

Fig. 2. Dependences for the loop of Example 2, when $n = 8$

wards paths from common iterations, contained in set *CI*, to sources of slices requiring synchronization.

Step 5*a* of Algorithm 1 can be applied to loops with both parameterized and non-parameterized bounds at compile or run-time whereas step 5 *b* can be applied to loops with parameterized bounds only at run-time when bounds become known.

It is well known that the exact transitive closure of an affine integer tuple relation may not be affine [24]. Exact transitive closure represented with affine forms is therefore not computable in the general case of the affine dependence relation. However, it is always computable for affine loops originating uniform dependences [24]. In the case when the Omega library computes inexact transitive closure, we may approximate this closure and try to extract slices.

To illustrate Algorithm 1 let us consider the following loop.

Example 2.

```
for(i = 1; i ≤ n; i++)
  for(j = 1; j ≤ n; j++) {
    a(2*i, 3*j) = b(i,j)
    b(i+1, j) = a(i, j)
}
```

The relations representing loop-carried dependences in this loop are as follows

(data dep.) R1 : {[i,j] → [2i,3j] : 1 <= j && 2i <= n && 1 <= i && 3j <= n},
(data dep.) R2 : {[i,j] → [i+1,j] : 1 <= i < n && 1 <= j <= n}.

Fig. 2 presents dependences for the loop of Example 2. Following Algorithm 1, we get the following results.

1. R := $R_1 \cup R_2$:= {[i,j] → [2i,3j] : 1 ≤ j && 2i ≤ n && 1 ≤ i && 3j ≤ n} union {[i,j] →
→ [i+1,j]: 1 ≤ i < n && 1 ≤ j ≤ n},
IR := inverse R := {[In_1,j] → [i,j′] : j = 3j′ && In_1 = 2i && 1 ≤ j′ && 2i ≤ n && 1 ≤ i && 3j′ ≤ n} union {[In_1,j] → [In_1−1,j] : 2 ≤ In_1 ≤ n && 1 ≤ j ≤ n},
CI:=EMPTY.
2. CI := {[i,j]: 1 ≤ j && 2i ≤ n && 1 ≤ i && 3j ≤ n} union {[i,j]: Exists (alpha,beta : j = =3alpha && 2beta = i && 2 ≤ i ≤ n && 3 ≤ j ≤ n)}.
3. I := domain R, J := range R,
UDS := I – J := {[1,j]: 1 ≤ j ≤ n && 2 ≤ n}.
4. Since *CI* is not EMPTY, go to step 5*a* (using transitive closure).
5. TIR := IR+ union {[i,j] → [i,j]} := {[2,j] → [1,j] : n = 2 && 1 ≤ j ≤ 2} union {[i,j] → [i,j] } union {[i,j] → [i′,j′] : 1 ≤ j′ ≤ j ≤ n && i′+2 ≤ i ≤ n && 1 ≤ i′ && 3j ≤ 2n+3j′ && 2i ≤ 4+n+2i′ && UNKNOWN} union ...

Since relation *TIR* includes UNKNOWN in the constraints, we approximate *TIR* to calculate the upper and lower bounds for *TIR*. Calculating and using the upper bound (UNKNOWN=True) for forming set *SFS* with Algorithm 1 yield the following result

SFS := {[1,j]: 2 ≤ n ≤ 3, 3j−1 && j ≤ 2} union {[1,j]: Exists (alpha : 4, j ≤ n ≤ ≤3alpha+2 && 3alpha < j)}.

Set *SFS* above does not include all sources of synchronization-free slices, we miss some sources.

Calculating the lower bound for *TIR*(UNKNOWN=False), *LTIR*,

LTIR := lower_bound TIR := {[2,j] → [1,j] : n = 2 && 1 ≤ j ≤ 2} union {[i,j] → [i,j] } union {[i,j] → [i′,j′] : j = 3j′ && 2i′ ≤ i ≤ n && 3j′ ≤ n && 1 ≤ j′ && 1 ≤ i′} union {[i,j] → → [i′,j′] : j = 9j′ && 4i′ ≤ i ≤ n && 9j′ ≤ n && 1 ≤ j′ && 1 ≤ i′} union {[i,j] → [i′,j] : 1 ≤ i′ < i≤ ≤ n && 1 ≤ j ≤ n && 3 ≤ n}

permits us to extract all sources of synchronization-free slices being contained in the following set

SFS := {[1,j]: Exists (alpha : j, 2 ≤ n ≤ 3j−1 && 3alpha+1 ≤ j ≤ 3alpha+2)}.

Finding subspaces in the loop domain where synchronization-free slices can be extracted. When synchronization-free slices cannot be extracted in the whole loop domain, we can try to find subspaces in the loop domain where synchronization-free slices can be extracted. The following algorithm analyzes dependence vectors in a particular way to discover subspaces of interest.

Algorithm 2. ***Finding subspaces with synchronization-free slices in the loop domain***.

Input: a source loop; set S_In being comprised of relations representing loop-carried dependences R_1, R_2, ..., R_m and set D being comprised of correspondent dependence vectors D_1, D_2, ..., D_m, where m is the number of relations.

Output: loop indices defining subspaces where the algorithm extracts sources of synchronization-free slices; set S_Out including dependence relations to be used for extracting slices in subspaces; sources of synchronization-free slices described with relations contained in set S_Out.

1. Check whether in set D there exists a dependence vector with one or more zero coordinates. If no, then the end, the algorithm does not extract any subspace with synchronization-free slices; otherwise go to step 2.

2. Set S_Out:= EMPTY.

2.1. Add to set S_Out dependence relation R_i belonging to set S_In and such that it has the minimal number of subsequent zero coordinates in the correspondent vector D_i among all the vectors in set D (if there exist two or more such relations, choose any of them; if a chosen relation yields in a correspondent dependence vector two or more sequences with the same number of zeros, chose any of them),

2.2. Add to set S_Out all the dependence relations from S_In such that the correspondent dependence vectors have the same zero coordinates as those chosen in D_i and the rest coordinates are arbitrary expressions (zero or nonzero),

2.3. Check whether set S_Out includes the same number of dependence relations that set S_In does. If so, the end, the algorithm does not extract any subspace with synchronization-free slices; otherwise go to step 3.

3. Apply Algorithm 1 to set S_Out. If it extracts sources of synchronization-free slices for S_Out, then check whether the source loop is semantically the same as that whose body is the same as that of the source loop and whose outer and inner loops are represented with indices that form the minimal number of subsequent zero coordinates of D_i and the rest ones, respectively (checking the legality of the loop interchange transformation); any known technique can be used for this purpose, for example, ones described in [1]. Else go to step 4.

If so, memorize the loop indices representing zeros in the chosen subsequence of zero coordinates in vector D_i; memorize the sources of synchronization-free slices for S_Out and set S_Out, the end. Else go to step 4.

4. Check whether in set D there exists a dependence vector, D_i, with another non-analyzed subsequence of zero coordinates. If so, S_Out:= EMPTY, add the correspondent dependence relation R_i into set S_Out and go to step 2.2, otherwise the end, the algorithm does not extract any subspace with synchronization-free slices.

It is worth to note that choosing in vector D_i the minimal number of subsequent zero coordinates aims at extracting the maximal number of synchronization-free slices. Dependence relations not included in set S_Out do not describe any dependence in each subspace of interest because of non-zero coordinates of correspondent dependence vectors derived from these relations. Each depend-

ence represented with relations not being included in set S_Out has a source and destination belonging to different subspaces (the source and destination of a dependence do not belong to the same subspace). Hence, we do not take them into account to extract sources of synchronization-free slices in appropriate subspaces of the loop domain. At generating code, we have to guarantee enumerating subspaces serially in lexicographic order to preserve dependences represented with relations not being included in set S_Out.

The following example illustrates applying Algorithm 2.

Example 3.

```
for (i = 1; i ≤ n; i++)
  for (j = 1; j ≤ n; j++)
  for (k = 1; k ≤ n; k++){
    a (i,j,k) = a (i,j–1,k)
    b (i,j,k) = b (i,j,k–1)
}
```

Dependence relations and correspondent dependence vectors for this loop are as follows.

R_1 := {[i,j,k] →[i,j+1,k] : 1 ≤ i ≤ n && 1 ≤ j < n && 1 ≤ k ≤ n},

D_1 := {[0,1,0]},

R_2 := {[i,j,k] → [i,j,k+1] : 1 ≤ i ≤ n && 1 ≤ j ≤ n && 1 ≤ k < n}.

D_2 := {[0,0,1]}.

The input date for this loop are sets $S = \{ R_1, R_2 \}$ and $D = \{ D_1, D_2 \}$. Applying Algorithm 2, we yield.

1. In set D there exists a dependence vector with zero coordinates, we go to step 2.

2. S_Out: = EMPTY.

2.1. We choose index k representing the zero coordinate in vector D_1 and form set S_Out := $\{ R_1 \}$.

2.2. S_Out: = $\{ R_1 \}$.

2.3. Because S_Out does not include the same number of dependence relations as S_In does, we go to step 3.

3. Applying Algorithm 1 to set S_Out we get the following sources of synchronization-free slices

SFS: = { [i,j,k] :j = 1 && 1 ≤ i ≤ n && 1 ≤ k ≤ n && 2 ≤ n }.

3.1. Because the correspondent loop interchange transformation is legal, we memorize index k, set *SFS*, and set S_Out.

Related work. The results of the paper are within the iteration space slicing framework introduced by Pugh and Rosser in paper [23]. This framework might have a number of uses. Pugh and Rosser examined in paper [23] how to apply

this framework to optimization of interprocessor communication. In particular, they demonstrated how to use slicing to enable loop fusion, tolerate message latency 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 [23] how to construct synchronization-free slices. Our contribution to the iteration space slicing framework consists in presenting how to extract sources of both synchronization-free slices and slices requiring synchronization. We are unaware of any work describing techniques exposing sources of synchronization-free slices.

The affine partitioning framework, considered in many papers, for example, [10 —12, 18, 19, 25 — 27] 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 the algorithms presented in this paper opens a possibility to extract synchronization-free slices when the affine partitioning framework and other well-known techniques can fail in extracting such slices.

The idea to seek for parallelism in subspaces of the loop domain is not new. It is discussed in many papers. Our contribution consists in demonstrating how Algorithm 1, presented in this paper, can be applied to subspaces of the loop domain and how these subspaces can be extracted.

Conclusion. In this paper, we described algorithms, permitting us to find sources of synchronization-free slices comprised of iterations of perfectly nested uniform and non-uniform loops. Extracting sources of synchronization-free slices will allow us to find iterations belonging to each synchronization-free slice. Each slice can be executed without synchronization with the other slices, thus allowing us to 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.

In our future work, using the results of this paper we intend to present how to extract iterations belonging to each synchronization-free slice and how to generate code enumerating sources of slices and iterations of each slice in lexicographic order.

Наведено алгоритми, що дозволяють знаходити несинхронізовані фрагменти, які вміщують ітерації повністю вкладених однорідних і неоднорідних циклів. Такі фрагменти можуть виконуватись незалежно, зберігаючи лексикографічний порядок ітерацій у кожному фрагменті. Запропонований підхід базується на операціях відношень та множин і потребує точного аналізу залежності між операторами програми. Для опису та реалізації алгоритмів обрано аналіз залежності по Пугу і Воннакоту, згідно з яким залежності знаходять у формі відношень кортежу. Описані алгоритми реалізовано і верифіковано за допомогою програмного пакета Omega project.

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. — Vol. 2658. — P. 925—934.
6. *Gavalda R., Ayguade E., Torres J.* Obtaining Synchronization-Free Code with Maximum Parallelism// Technical Report LSI-96-23-R. — Barcelona: Universitat PolitPcnica 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.—No 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 13th 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 24th 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. *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.
15. *Beletskyy V., Siedlecki K.* Finding Free Schedules for Non-uniform Loops// Proc. of the Euro-Par 2003. Lecture Notes in Computer Science.— Berlin/Heidelberg: Springer, 2003. Vol. 2790. — P. 297—302.
16. *Boulet P., Darte A., Silber G.A., Vivien F.* Loop parallelization algorithms: from parallelism extraction to code generation// Parallel Computing, 1998.—No 24.— P. 421—444.

17. *Collard J. F., Feautrier P., Risset T.* Construction of do loops from systems of afffne constraints. Construction of Do Loops from Systems of Affine Constraints//Parallel Processing Letters. — 1995. — P. 421—436.
18. *Feautrier P.* Some efficient solutions to the affine scheduling problem, part i, one dimensional time// International J. of Parallel Programming. — 1992. — Vol. 21, No5.— P. 313—348.
19. *Feautrier P.* Some efficient solutions to the affine scheduling problem, part ii, multidimensional time// Ibid. — P. 389— 420.
20. *Pugh W., Wonnacott D.* Constraint-based array dependence analysis//ACM Trans. on Programming Languages and Systems. — 1998.— Vol. 20, No3. — 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, No7. — P. 352—357.
23. *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.
24. *Kelly W., Pugh W., Rosser E., Shpeisman T*. Transitive Closure of Infinite Graphs and its Applications// Intern. J. of Parallel Programming. — 1996. — V. 24, No 6. — P. 579—598.
25. *Darte A., Robert Y., Vivien F.* Scheduling and Automatic Parallelization. — Boston : Birkhäuser, 2000. — 294 p.
26. *Feautrier P.* Toward automatic distribution//J. of Parallel Processing Letters. — 1994. — Vol. 3, No4. — P. 233—244.
27. *Quillere F., Rajopadhye S., Wilde D.* Generation of efficient nested loops from polyhedra// International J. of Parallel Programming. — 2000. — Vol. 28, No5. — P. 469—498.

Поступила 07.11.06;
после доработки 26.02.07