Włodzimierz BIELECKI [1], **Marek PALKOWSKI** [1], **Anna BELETKSA** [2]
[1] WEST POMERANIAN UNIVERSITY OF TECHNOLOGY, DEPARTMENT OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY
[2] INRIA SACLAY, FRANCE

Extracting representative loop statement instances of synchronization-free slices

Prof. dr hab. inż. Włodzimierz BIELECKI

Prof. dr hab. inż. Włodzimierz Bielecki is head of the Software Technology Department of the West Pomeranian University of Technology, Szczecin. His research interest includes parallel and distributed computing, optimizing compilers, extracting both fine- and coarse grained parallelism available in program loops.

e-mail: wbielecki@wi.zut.edu.pl

Dr inż. Marek PAŁKOWSKI

Obtained his PhD degree in Computer Science from the Technical University of Szczecin, Poland. The main goal of his research is extracting parallelism from program loops and developing Iteration Space Slicing Framework.

e-mail: mpalkowski@wi.zut.edu.pl

Abstract

Extracting synchronization-free parallelism by means of the Iteration Space Slicing Framework consists of two steps. First, representative loop statement instances of slices are extracted. Next, slices are reconstructed from their representatives and parallel code scanning slices and elements of each slice is generated. In this paper, we present how to benefit from this technique in practice. We explain how to extract representative loop statement instances of slices by means of the Omega Library enlarged by four new functions allowing us to simplify the process of extracting slice representatives. Results of experiments with the NAS and UTDSP benchmarks are presented.

Keywords: synchronization-free slices, parallelism, representative loop statement instances.

Ekstrakcja instancji instrukcji fragmentów kodu pozbawionych synchronizacji w pętlach programowych

Streszczenie

Rozwój architektur wielordzeniowych wymusza poszukiwanie algorytmów automatycznego zrównoleglenia aplikacji. W artykule opisano zrównoleglenie pętli programowych za pomocą ekstrakcji niezależnych fragmentów kodu. Ekstrakcja równoległości w pętlach programowych pozbawionych synchronizacji za pomocą podziału przestrzeni iteracji składa się z dwóch kroków. Najpierw znajdowane są instancje instrukcji będące początkami fragmentów kodu. Następnie fragmenty kodu uzupełniane są o wszystkie instrukcje i generowany jest kod równoległy. W artykule przedstawiono korzyści wynikające z takiego podejścia. Wyjaśniono sposób poszukiwania instancji instrukcji fragmentów kodu za pomocą biblioteki Omega rozszerzonej o nowe funkcje upraszczające poszukiwanie instrukcji należących do fragmentów kodu. Opis proponowanego podejścia uzupełniono o zbiór eksperymentów na pętlach testowych NAS i UTDSP.

Słowa kluczowe: fragmenty kodu pozbawione synchronizacji, równoległość, instancje instrukcji pętli programowych.

Dr inż. Anna BELETSKA

She is graduated from the Technical University of Szczecin in 2004, and has obtained her PhD degree in Computer Science from Politecnico di Milano in 2008. Currently she is carrying out Post Doctoral research in INRIA, France, developing advanced coarse-grained parallelization and code generation techniques to be embedded into the GCC compiler. The main research goal is to devise novel techniques for calculating the transitive closure of a union of dependence relations and apply them for extracting synchronization-free slices.

e-mail: anna.beletska@inria.fr

1. Introduction

In our recent work [2, 3, 4] we have proposed several algorithms to extract coarse-grained parallelism represented with synchronization-free slices consisting of the loop statement instances by means of the Iteration Space Slicing Framework (ISSF). The goal of those publications was the formalization and theoretical representation of techniques extracting synchronization-free slices by means of ISSF. Experimental results with NAS benchmarks (the number of slices extracted from loops as well as program performance: speedup and efficiency of parallel code) are presented in paper [2].

In the current paper, we focus on the problem of the automatic extraction of representative loop statement instances of synchronization-free slices in practice, because this problem is not trivial and requires a special attention.

In order to implement our algorithms, we have chosen the publically available Omega Library [11]. Although Omega provides many useful high-level functions, it lacks some very important functions to be able to generate specialized relations envisaged by our algorithms.

In this paper, we show how to extract loop statement instances of synchronization-free slices by means of both available functions of the Omega Library and new additional functions which we have implemented.

2. Background

In this paper, we deal with affine loop nests where:

i) 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 (i.e., parameterized loop bounds), and
ii) the loop steps are known positive constants.

A nested loop is called perfectly nested if all its statements are comprised within the innermost nest. Otherwise, the loop is called imperfectly nested. An arbitrarily nested loop can be both perfectly and imperfectly nested.

A statement instance $s(I)$ is a particular execution of a loop statement s for a given iteration I.

Two statement instances $s_1(I)$ and $s_2(J)$ are *dependent* if both access the same memory location and if at least one access is a write. $s_1(I)$ and $s_2(J)$ are called the source and destination of a dependence, respectively, provided that $s1(I)$ is lexicographically smaller than $s_2(J)$ ($s_1(I) \prec s_2(J)$, i.e., $s_1(I)$ is always executed before $s_2(J)$).

The approach to extract synchronization-free parallelism in program loops by means of the Iteration Space Slicing Framework requires an exact representation of loop-carried dependences and consequently an exact dependence analysis which detects

a dependence if and only if it actually exists[1]. To describe and implement our algorithms, we choose the dependence analysis proposed by Pugh and Wonnacott [8] where dependences are represented by dependence relations whose constraints are described in the Presburger arithmetic (built of affine equalities and inequalities, logical and existential operators); the Omega library is used for computations over such relations [11].

A dependence relation is a tuple relation of the form

$$\{[input\ list] \rightarrow [output\ list] : constraints\}; \quad (1)$$

where *input list* and *output list* are the lists of variables and/or expressions used to describe input and output tuples and *constraints* is a Presburger formula describing 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)), relation application (given a relation R and set S, $R(S) = \{[e']: \exists e \in S,\ e \rightarrow e' \in R\}$), positive transitive closure (given a relation R, $R^+ = \{[e] \rightarrow [e']: e \rightarrow e' \in R \ ||\ \exists e''\ s.t.\ e \rightarrow e'' \in R\ \&\ e'' \rightarrow e' \in R^+\}$), transitive closure ($R^* = R^+ \cup I$, where I is the identity relation). These operations are described in detail in [11].

Iteration Space Slicing [7] takes dependence information as input to find all statement instances that must be executed to produce the correct values for the specified array elements.

Definition 1

Given a dependence graph, D, defined by a set of dependence relations, S, a *slice* is a weakly connected component of graph D, i.e., a maximal subgraph of D 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 D, then taking into account the above definition, we may conclude that all slices are synchronization-free, i.e., there is no dependence between them.

Definition 2

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

Definition 3

A *source(s) of a slice* is an ultimate dependence source(s) that this slice contains.

Definition 4

A *representative* loop statement instance of a slice is its lexicographically minimal source.

Further on in this paper, we refer to representative loop statement instances as to representatives.

3. Extracting representatives of slices

The approach to extract synchronization-free slices [2] relies on the transitive closure of an affine dependence relation describing all dependences in a loop and consists of two steps. First, representatives of slices are found in such a manner that each slice is represented with its lexicographically minimal statement instance. Next, slices are reconstructed from their representatives and code scanning these slices is generated.

Given a dependence relation R describing all dependences in a loop, we can find a set of statement instances, S_{UDS}, describing all ultimate dependence sources of slices as S_{UDS}=domain(R) - range(R). In order to find elements of S_{UDS} that are representatives of slices, we build a relation, R_{USC}, that describes all pairs of the ultimate dependence sources that are transitively connected in a slice, as follows:

$$R_{USC} := \{[e] \rightarrow [e'] : e, e' \in S_{UDS}, e \prec e', R^*(e') \cap R^*(e)\}, \quad (2)$$

where R^* is the transitive closure of relation R.

The condition ($e \prec e'$) in the constraints of relation R_{USC} means that e is lexicographically smaller than e′. Such a condition guarantees that the lexicographically smallest element from e and e' will always appear in the input tuple, i.e., the lexicographically smallest source of a slice (its representative source) can never appear in the output tuple. The intersection $R^*(e') \cap R^*(e)$ in the constraints of R_{USC} guarantees that elements e and e' are transitively connected, i.e., they are the sources of the same slice.

Set S_{repr} containing representatives of each slice is found as $S_{repr} = S_{UDS}$- range(R_{USC}).

Each element e of set S_{repr} is the lexicographically minimal statement instance of a synchronization-free slice. If e is the representative of a slice with multiple sources, then the remaining sources of this slice can be found applying relation (R_{USC})* to e, i.e., (R_{USC})*(e). If a slice has the only source, then (R_{USC})*(e)=e. The elements of a slice represented with e can be found applying relation R^* to the set of sources of this slice: $S_{slice} = R^*((R_{USC})^*(e))$.

4. Enlarging the Omega Library

The Omega Library [5, 11] is a set of C++ classes for manipulating integer tuple relations and sets. Sets and relations in the Omega library are implemented as a single class, *Relation*, marked as being either a set or a relation. A Presburger formula describing the constraints imposed upon input and output tuples of a relation is represented as a tree of formula nodes that can have zero or more children. Children can be either other formula nodes or "atomic" constraints (single equality, inequality, or stride constraints). *Formula* is an abstract class and a plain *Formula* node can never be used in a tree. The subclasses of formulas are *F_And, F_Or, F_Not, F_Declaration, F_Forall, F_Exists*. Children can be added to any subclass using the following member functions that return pointers to the newly created child node: *F_And* Formula::add_and(), F_Or* Formula::add_or(), F_Exists* Formula::add_exists(), etc*.

In order to be able to represent the constraint ($R^*(e') \cap R^*(e)$) of relation R_{USC} in Omega, we redefine it as an equivalent constraint of the form:

$$(\exists e'': e'' \in D\ \&\ e'' \neq e\ \&\ e'' \neq e'\ \&\ e'' \in R^*(e)\ \&\ e'' \in R^*(e')) \quad (3)$$

where $D = Domain(R) \cup Range(R)$, meaning that two sources of a slice, e and e', are transitively connected if there exists such an element, e'', in the space of dependent statement instances D that belongs to both $R^*(e)$ and $R^*(e')$.

None of the constraints of relation R_{USC} can be represented in Omega directly. That is why, to permit for forming the constraints of relation R_{USC}, we implemented the following functions.

1. ***void S_Add_Constraint (Set S, Relation R, F_And *fex_and, Variable_ID ex[ ])*** which substitutes the variables of set S for the variables represented by *ex*[ ] and inserts the constraints represented by S into the constraints of relation R.

[1] A non-exact representation of dependences is also possible, but this will cause losses in some parallelism because of the over-approximation of dependences, while we aims at extracting maximal synchronization-free parallelism.

2. ***void R_Add_Constraint (Relation R1, Relation R2, F_And *fex_and, Variable_ID ex[ ], int flag)*** which substitutes the variables of the input or output tuple of relation *R*1 or the variables represented by *ex*[ ] and inserts the constraints of relation *R1* into the constraints of relation *R*2.

3. ***void Add_Lexicographic_Order (Relation R, F_And *fex_and)***, or ***void Add_Lexicographic_Order (Relation R, F_And *fex_and, Tuple<Variable_ID> left, Tuple <Variable_ID> right)*** which inserts the lexicographic order constraint into the constraints of relation *R*.

4. ***void Diff_Add Constraint (Relation R, F_And *fex and, Variable_ID ex[ ])*** which inserts constraint "*input_tuple≠tuple* & *output_tuple≠tuple*", where "*tuple*" is defined by *ex[ ]*, into the constraints of relation R.

Tab. 1. Generation of R_{USC}
Tab. 1. Obliczanie R_{USC}

Step	Constraints	Code
1	$R\_USC = \{[e] \rightarrow [e']\}$	Relation R_USC(n,n);
2	$e,e' \in UDS$	S_Add_Constraint(UDS, R_USC, fand, e); S_Add_Constraint(UDS, R_USC, fand, e');
3	$e \prec e'$	Add_Lexicographic Order(R_USC, fand);
4	$\exists e'': e'' \in D$	S_Add_Constraint(D, R_USC, fexand, e''),
5	$e'' \neq e \,\&\, e'' \neq e'$	Diff_Add_Constraint(R_USC, fexand, e'');
6	$e'' \in R^*(e)\,\&$ $e'' \in R^*(e'))$	R_Add_Constraint(R*, R_USC, fexand, e'', 1); R_Add_Constraint(R*, R_USC, fexand, e'', 0);

The library of the above functions with their detailed description can be found in the section "Download" at http://detox.wi.ps.pl/SFS_Project. Table 1 demonstrates how relation R_{USC} can be formed in 6 steps by means of the implemented functions. As we can see, using the implemented functions, we can generate R_{USC} by the 8 lines of transparent code. Having R_{USC} generated, it is important to simplify it using Omega's *simplify* function.

5. Experiments

The presented new functions were used in the tool, ESyS, implemented by us and permitting for automatic extracting representative loop statement instances of slices and generating code scanning slices and elements of each slice. It can be downloaded from http://detox.wi.ps.pl/SFS_Project. Using ESyS, we have extracted representative loop statement instances of slices for the loops presented in Table 2. The graphical representations of slices available in the loops are presented in Figures 1 – 4. For loops 1, 2, and 3 the considered iteration space is the set $\{[i,j]: 1 \le i \le 5 \,\&\, 1 \le j \le 4\}$, while for loop 4 it is the set $\{[i,j]: 1 \le i,j \le 6\}$. The representatives of slices are marked with the black circles.

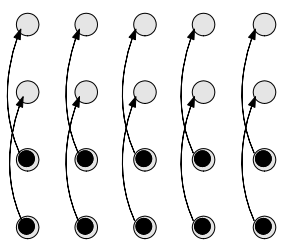

Fig. 1. The dep. graph for loop 1
Rys. 1. Graf zależności dla pętli 1

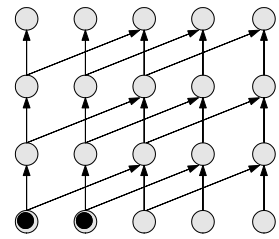

Fig. 2. The dep. graph for loop 3
Rys. 2. Graf zależności dla pętli 3

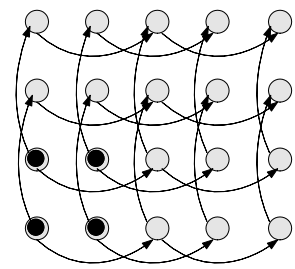

Fig. 3. The dep. graph for loop 2
Rys. 3. Graf zależności dla pętli 2

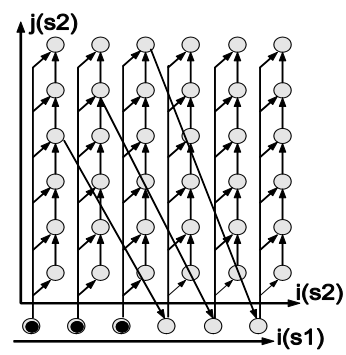

Fig. 4. The dep. graph for loop 4
Rys. 4. Graf zależności dla pętli 4

Tab. 2. Extracting representative loop statement instances of slices
Tab. 2. Ekstrakcja instancji instrukcji fragmentów kodu w pętli

Loop 1.
for (i=1; i<=n; i++)
for (j=1; j<=n; j++)
a[i][j]=a[i][j+2];
 	
$R_{USC} = \phi$;
S_{repr}={[i][j] 1 ≤ j ≤ n-2, 2 & 1 ≤ i ≤ n}.
	Loop 2.
	for (i=1; i<=n; i++)
	for (j=1; j<=n; j++)
	a[i][j]=a[i][j+2]+a[i+2][j];
	
	$R_{USC} = \phi$;
	S_{repr}= {[i,j]: 1 ≤ j ≤ n-2, 2 & 1 ≤ i ≤ n-2, 2}.

Loop 3.
for (i=1; i<=n; i++)
for (j=1; j<=n; j++)
a[i][j]=a[i][j+1]+a[i+2][j+1];
 	
R_{USC}= {[i,1] → [i',1]:
∃ (alpha: 0=i+i'+ 2alpha &
1 ≤ i ≤ i' ≤2 & i' ≤n)};
S_{repr}= {[i,1]: 1≤i≤2 & 2≤n}.
	Loop 4.
	for (i=1; i<=n; i++){
	s1: b[i][i]=a[i-3][i];
	for (j=1; j<=n; j++)
	s2: a[i][j]=a[i][j-1] +b[i][i];
	}
	
	$R_{USC} = \phi$;
	S_{repr}= {s2 [i]: 1≤i≤n,3}.

Applying ESyS, we have studied NAS [10] and UTDSP[12] benchmarks to recognize what is the number and percentage of loops exposing multiple synchronization-free slices. We have

considered only such loops for which Petit [11] was able to carry out a dependence analysis. Table 3 presents the number and percentage of loops for which multiple slices were extracted and those of loops for which only a single slice was extracted. From the results presented in Table 3, we may conclude that for most loops from examined benchmarks, the Iteration Space Slicing Framework is able to extract coarse-grained parallelism.

Tab. 3. Results of experiments
Tab. 3. Wyniki dla zbiorów pętli testowych

Benchmark	The number of loops	Loops with multiple slices		Loops with a single slice	
NAS	80	69	86.25%	11	13.75%
UTDSP	21	16	76.2%	5	23.8%

6. Related work

The results of the paper are within the Iteration Space Slicing Framework (ISSF) introduced by Pugh and Rosser [7]. That paper examines one of possible uses of ISSF, namely how to optimize interprocessor communication. However, Pugh and Rosser do not show how synchronization-free slices can be extracted. Our previous papers [3, 4] present approaches within ISSF to extract synchronization-free slices of the chain and tree topologies that are described with non-linear forms. Because slices of the chain and tree topology always have a single source, there is no need for calculating relation R_{USC} and the problem of the computation of representative loop statement instances does not arise. Paper [2] is also within ISSF. It is devoted to the extraction of synchronization-free slices described with affine forms, but it does not present any details concerning the implementation of a proposed way to expose representative loop statement instances of slices.

7. Conclusion and future work

We presented a way how to calculate representative loop statement instances of synchronization-free slices in practice by means of the enlarged Omega library. Four additional functions were added to the library to simplify the implementation of extracting synchronization-free slices. Experiments with NAS and UTDPS benchmarks were carried out by means of a tool implementing the presented approach. We extracted representatives of synchronization-free slices for NAS and UTDPS loops for which Petit was able to carry out a dependence analysis. In our future work we plan to examine more popular benchmarks to discover what is the percentage of loops exposing synchronization-free slices.

8. References

[1] Bastoul C.: Code generation in the polyhedral model is easier than you think. In PACT'2004, pp. 7-16, Juan-les-Pins, September 2004.
[2] Beletska A., Bielecki W., Siedlecki K., and San Pietro P.: Finding synchronization-free slices of operations in arbitrarily nested loops. In ICCSA (2), volume 5073 of Lecture Notes in Computer Science, pp. 871-886. Springer, 2008.
[3] Bielecki W., Beletska A., Palkowski M., and San Pietro P.: Extracting synchronization-free chains of dependent iterations in non-uniform loops. In ACS '07: Proceedings of International Conference on Advanced Computer Systems, 2007.
[4] Bielecki W., Beletska A., Palkowski M., and San Pietro P.: Finding synchronization-free parallelism represented with trees of dependent operations. In ICA3PP, volume 5022 of LNCS, pp.185-195. Springer, 2008.
[5] Kelly W., Maslov V., Pugh W., Rosser E., Shpeisman T., and Wonnacott D.: The omega library interface guide. Technical report, USA, 1995.
[6] Lim A. W., Cheong G. I., Lam M. S.: An affine partitioning algorithm to maximize parallelism and minimize communication. In ICS'99, pp.228-237. ACM Press, 1999.
[7] Pugh W. and Rosser E.: Iteration space slicing and its application to communication optimization. In International Conference on Supercomputing, pp. 221-228, 1997.
[8] Pugh W. and Wonnacott D.: An exact method for analysis of value-based array data dependences. In In Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing. Springer-Verlag, 1993.
[9] Vasilache N., Bastoul C., Cohen A..: Polyhedral code generation in the real world. In ETAPS CC'06), LNCS 3923, pp 185-201, Vienna, Austria, March 2006. Springer-Verlag.
[10] NAS benchmark suite. http://www.nas.nasa.gov.
[11] The Omega project. http://www.cs.umd.edu/projects/omega.
[12] UTDSP benchmark suite. http://eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html.

Artykuł recenzowany