Extracting Both Affine and Non-linear Synchronization-Free Slices in Program Loops

Wlodzimierz Bielecki and Marek Palkowski

Faculty of Computer Science, Technical University of Szczecin,
70210, Zolnierska 49, Szczecin, Poland
{bielecki,mpalkowski}@wi.ps.pl
http://kio.wi.zut.edu.pl/

Abstract. An approach is presented permitting for extracting both affine and non-linear synchronization-free slices in program loops. It requires an exact dependence analysis. To describe and implement the approach, the dependence analysis by Pugh and Wonnacott was chosen where dependences are found in the form of tuple relations. The approach is based on operations on integer tuple relations and sets and it has been implemented and verified by means of the Omega project software. Results of experiments with the UTDSP benchmark suite are discussed. Speed-up and efficiency of parallel code produced by means of the approach is studied.

Keywords: synchronization-free slices, free-scheduling, parallel shared memory programs, dependence graph.

1 Introduction

Microprocessors with multiple execution cores on a single chip are the radical transformation taking place in the way that modern computing platforms are being designed. Hardware industry is moving toward the direction of multi-core programming and parallel computing. This opens new opportunities for software developers [1].

Multithreaded coarse-grained applications allow the developer to exploit the capabilities provided by the underlying parallel hardware platform including multi-core processors. Coarse-grained parallelism is obtained by creating a thread of computations on each execution core (or processor) to be executed independently or with occasional synchronization.

Different techniques have been developed to extract synchronization-free parallelism available in loops, for example, those presented in papers [2,3,4,5,6]. Unfortunately, none of those techniques extracts synchronization-free parallelism when the constraints of sets representing slices are non-linear.

In our recent work [7,8], we have proposed algorithms to extract coarse-grained parallelism represented with synchronization-free slices described by non-linear constraints. But they are restricted to extracting slices being represented by a graph of the chain or tree topology only.

R. Wyrzykowski et al. (Eds.): PPAM 2009, Part I, LNCS 6067, pp. 196–205, 2010.

The main purpose of this paper is to present an approach that permits us to extract synchronization-free slices when they are represented

i) by both affine and non-linerar constraints,
ii) not only by the chain or tree topology but also by a graph of an arbitrarily topology.

The presented algorithm is applicable to the parameterized both uniform and non-uniform arbitrary nested loop.

2 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 constants.

Two statement instances I and J are *dependent* if both access the same memory location and if at least one access is a write. I and J are called the *source* and *destination* of a dependence, respectively, provided that I is lexicographically smaller than J ($I \prec J$, i.e., I is always executed before 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 actually exists. To describe and implement our algorithm, we chose the dependence analysis proposed by Pugh and Wonnacott [11] where dependences are represented by dependence relations.

A dependence relation is a tuple relation of the form [*input list*]→[*output list*]: *formula*, where *input list* and *output list* are the lists of variables and/or expressions used to describe input and output tuples and *formula* describes the constraints imposed upon *input list* and *output list* and it is a Presburger formula built of constraints represented with algebraic expressions and using logical and existential operators.

We use standard operations on relations and sets, such as intersection ($\cap$), union ($\cup$), difference (-), domain (dom R), range (ran R), relation application ($S' = R(S)$: $e' \in S'$ iff exists e s.t. $e \rightarrow e' \in R, e \in S$), transitive closure of relation R, R*. In detail, the description of these operations is presented in [11,12,13].

Definition 1. An *iteration space* is a set of all statement instances that are executed by a loop.

Definition 2. An *ultimate dependence source* is the source that is not the destination of another dependence.

Defnition 3. 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.

Definition 4. Given a relation R, *set of common dependence sources*, CDS, is defined as follows

CDS := {[e] : Exists(e',e'', s.t., e = $R^{-1}(e') = R^{-1}(e'')$ && e',e''∈ range(R) && e'≠e'') }.

Definition 5. Given a relation R, *set of common dependence destinations*, CDD, is defined as follows
CDD := {[e] : Exists(e',e''s.t., e = R(e') = R(e'') && e',e''∈ domain(R) && e'≠e'')}.

We distinguish the following three kinds of the dependence graph topology

i) chain when CDS = ϕ and CDD = ϕ,
ii) tree when CDS ≠ ϕ and CDD = ϕ,
iii) multiple incoming edges graph - MIE graph when CDD ≠ ϕ.

A *schedule* is a mapping that assigns a time of execution to each statement instance of the loop in such a way that all dependences are preserved, that is, mapping s:I→Z such that for any dependent statement instances si1 and si2 ∈ I, s(si1)<s(si2) if si1 ≺ si2 [9].

Definition 7 [9]. A *free schedule* assigns statement instances as soon as their operands are available, that is, mapping σ:I→Z such that

$$\sigma(p) = \begin{cases} 0 \; if \; thereis \; no \; p' \in I \; s.t. \; p' \prec p \\ 1 + max(\sigma(p')), p' \in I, p' \prec p \; . \end{cases}$$

The free schedule is the "fastest" schedule possible. Its total execution time is

$$T_{free} = 1 + max(\sigma_{free}(p), \; p \in I) \; .$$

3 Motivating Example

To explain troubles we deal with non-linear slices, let us consider the following example

```
for i = 1 to n
 for j = 1 to n
  a(i,j) = a(2*i,j) : a(i,j-1)
 endfor
endfor
```

Dependences in this loop are represented by the following relations
R_1:=[i,j] → [2i,j]: 1⩽i && 2i ⩽ n && 1 ⩽ j < n ,
R_2:=[i,j] → [i,j+1] : 1 ⩽ i ⩽n && 1 ⩽ j < n .

From Figure 1, illustrating dependences in the loop for n=6, we can see that there exist 3 synchronization-free slices represented by the following sets of statement instances

$$s_k := \{[i,j] \; : \; 1 \le i \le n \; \&\& \; 1 \le j \le n \; \&\& \; Exists \; \alpha \; : \; i = 2^{\alpha} * z_k\},$$

where k=1,2,3 and z_1=1, z_2=3, z_3=5. The constraints of the sets above are non-linear. We are unaware of any technique permitting for generating code for

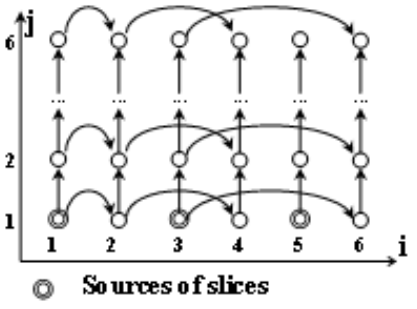

Fig. 1. Iteration space and dependences for the motivating example, n=6

such sets. Our previous algorithms [7,8] allow one to parallelize only such loops whose dependence graphs are chains or trees. The dependence graph for the motivating example is a multiple incoming edge graph. Below we propose an algorithm permitting for generating code enumerating statement instances of both affine and non-linear slices.

4 Algorithm

The idea of the algorithm presented below is the following. First, using a set of dependence relations, we check whether the topology of a dependence graph is a MIE graph. If so, relations are modified so that they do not represent any common dependences destination. Then we check whether such modifications are permitted. If so, using modified relations, we generate code that preserves all dependences represented by the original relations.

Algorithm. Extracting both affine and non-linear synchronization-free slices.
Input: Set of dependence relations R_i, $1 \leqslant i \leqslant q$, S; set of slice sources, Sour, calculated by means of the technique presented in [7].
Output: code scanning synchronization-free slices when possible.

1 Using set S, calculate the union of relations $R := \bigcup_{1 \leq i \leq q} R_i$.
2 Using R, calculate set of common dependence destinations, CDD.
If CDD = ϕ then $S' = S$, go to step 5 */* satisfying this condition means that the dependence graph described by R is not a MIE graph */*
3 For each relation R_i, $1 \leqslant i \leqslant q$ from set S do
3.1 Form relation $R'_i := R_i = \{[e] \rightarrow [e'] : \text{constraints}\}$ and insert in its constraints the following constraint:
&& $e' \in$ range R_i && $e' \notin$ CDD, */* this constraint causes that R'_i does not describe any common dependence destination*/*
3.2 If the following condition $R_i - R'_i = \phi$ AND $R'_i - R_i = \phi$, */* satisfying this condition means that Ri and R'_i are the same */*
is not satisfied, then do

3.2.1 Create a new set of relations $S^{'}$, where R_i is replaced for $R^{'}{}_i$
3.2.2 Calculate $R^{'}$as the union of all relations contained in set $S^{'}$
3.2.3 If R+(Sour) -$R^{'}$+(Sour)= ϕ and $R^{'}$+(Sour)- R+(Sour)= ϕ, */*satisfying this condition means that the transitive closure of R and that of $R^{'}$ are the same */*
where R+ is the transitive closure of R, then replace R_i for $R^{'}{}_i$ in set S,

4 Using the modified set $S^{'}$, calculate the union of relations $R:= \bigcup_{1 \le i \le q} R_i$; using R, calculate set of common dependence destinations, CDD, and check whether CDD = ϕ; if not, the end, the algorithm fails to extract slices.
5 Generate parallel code for set of dependence relations $S^{'}$using the algorithm presented in [8].

Proof. To prove the correctness of the algorithm above, we should demonstrate that the resulting code produced by the algorithm preserves all the dependences in the original loop. With this purpose, let us remind that code produced by the technique presented in paper [8] assigns statement instances to be executed according to free-scheduling. To show that the free scheduling is the same for both a set of modified relations and a set of original relations, we observe that time k when a given statement instance, I, is executed according to free-scheduling, is defined by the number of vertices on the maximal path from the source of a slice, S, to I (a path in a graph is a sequence of vertices such that from each of its vertices there is an edge to the next vertex in the sequence). Because satisfying the condition

$$R + (Sour) - R' + (Sour) = \phi \; and \; R' + (Sour) - R + (Sour) = \phi$$

guarantees that for any vertex I, the maximal path from S to I is the same for the both the dependence graph formed by a set of original dependence relations and that formed by a set of modified relations produced by the algorithm, we can conclude that the free scheduling is the same for both the original dependence graph and the dependence graph produced by the algorithm, i.e., the resulting code, produced by the algorithm, preserves all the dependences in the original loop.

From the proof above, it is clear when the algorithm fails to produce parallel code. This takes place when the algorithm is not able to produce a non-MIE dependence graph or the free scheduling for a produced non-MIE graph is different from that produced for the original dependence graph.

In the motivating example, each ultimate dependence source is the source of a slice (the dependence graph does not contain any pair of ultimate dependence sources that is connected by the undirected path in the dependence graph). So, sources of slices can be calculated as Domain(R)-Range(R). In papers [7,8], we proposed algorithms to generate *while loop* based code scanning non-linear slices of the chain and /or tree topology. Each iteration of a while loop executes all the statement instances whose operands have already been calculated and prepares a set of statement instances to be executed at the next iteration. Statement

instances are executed as soon as their operands are available, preserving in such a way all dependences exposed for an original loop. Such a schedule (assigning a time of execution to each operation of the loop) is known to be free-schedule [9].

Below we demonstrate applying the algorithm to the motivating example. Set S of dependence relations is the following
R_1:={[i,j] → [2i,j]: 1⩽i && 2i ⩽ n && 1 ⩽ j < n },
R_2:={[i,j] → [i,j+1] : 1 ⩽ i ⩽n && 1 ⩽ j < n }.
A set of sources, Sour, is calculated by the algorithm presented in [7]
Sour := {[i,1]: Exists (alpha : 2alpha = 1+i && 1 ⩽ i ⩽ n, 2n-3}.

1 R :=R_1∪R_2={[i,1]→[2i,1]: 1⩽i && 2i⩽n && 1⩽j < n }∪{[i,j]→[i,j+1]:1⩽i⩽n &&1⩽j<n};
2 CDD:={[i,j] : Exists (alpha : 2alpha = i && 1⩽i⩽n && 2⩽j⩽n)} ≠ ϕ
3 For R_1:
3.1 R'_1 := {[i,1] → [2i,1]: 1⩽i && 2i⩽n },
3.2 R_1-R'_1 = {[i,j] → [2i,j]: 1⩽i && 2i⩽n && 2⩽j ⩽ n } ≠ ϕ
3.2.1 S'= R'_1, R_2
3.2.2 R'= {[i,1] → [2i,1]: 1⩽i && 2i⩽n}∪ {[i,j] → [i,j+1] : 1 ⩽ i ⩽n && 1 ⩽ j < n }
3.2.3 R+ = {[i,j] →[i',j']: Exists(k1,k2,k : k≥1 && k1+k2=k && i'= 2^k*i && j'= j+k2) && 1 ⩽i⩽n && 2i ⩽ n && 1⩽j<n && 1⩽i⩽n && (Exists(alpha: i'= 2*alpha && 2⩽i'⩽n, 2n && 2⩽j'⩽2n)∪1⩽i'⩽n && 2⩽j'⩽n)
R'+ = {[i,j] →[i',j']: Exists(k1,k2,k : k≥1 && k1+k2=k && i'= 2^k*i && j'= j+k2) && j=1 && 1 ⩽i⩽n && 2i ⩽ n && 1⩽i⩽n && (Exists(alpha: i'= 2*alpha && 2⩽i'⩽n, 2n && j'=1)∪1⩽i'⩽n && 2⩽j'⩽n)
R+(Sour) = R'+(Sour) = {[i,1] Exists(alpha : 2alpha = i && 1 ⩽i⩽n)} ∪ {[i,j] : 1 ⩽i⩽n && 2 ⩽j⩽n}
S := {R'_1,R_2} R:=R'

3 For R_2
3.1 R'_2 = { [i,j] → [i,j+1] = 1⩽i⩽n && 1⩽j⩽n && Exists(alpha : 2alpha = i+1 && 1 ⩽i⩽n, 2n-3)}
3.2 R_2 - R'_2 = { [i,j] → [i,j+1] = 1⩽i⩽n && 1⩽j⩽n && Exists(alpha : 2alpha = i && 1 ⩽i⩽n, 2n-3)} ≠ ϕ
3.2.1 S'= {R'_1, R'_2}
3.2.2 R'= {[i,1] → [2i,1]: 1⩽i && 2i⩽n }∪ { [i,j]→[i,j+1] = 1⩽i⩽n && 1⩽j⩽n && Exists(alpha : 2alpha = i+1 && 1 ⩽i⩽n, 2n-3)}
3.2.3 R'+ = {[i,j] → [i',j']: Exists(k1,k2,k : k≥1 && k1+k2=k && i'= 2^k*i && j'= j+k2) && j=1 && 1 ⩽i⩽n && 2i ⩽ n && 1⩽i⩽n && Exists(alpha : 2alpha = i+1 && 1 ⩽i⩽n, 2n-3) && (Exists(alpha: i'= 2*alpha && 2⩽i⩽n, 2n && j'=1)∪1⩽i'⩽n && 2⩽j'⩽n) && Exists(alpha : 2alpha = i'+1 && 1 ⩽i'⩽n, 2n-3)}

R+(Sour)-R$^{'}$+(Sour) = {[i,j] : 2 < j ⩽n && 1 ⩽i ⩽n && Exists(alpha : 2alpha = i)} $\neq \phi$ /* *transitive closure of R and that of R$^{'}$ are not the same* */
S = {R$^{'}_1$, R_2}

4 CDD = ϕ. The dependence graph is represented by the tree (Fig. 2).

5 Using the technique presented in [8], we generate the following parallel code.

```
parfor(t=1;t<=min(n,2*n-3);t+=2)
{
  I=[t,1];
  Add I to S';       /* I is the source of a tree */
    while(S'!= null) {
      S_tmp=null;
      (par)foreach(vector I=[i,j] in S') {
        s1(I); /* the original loop statement */
               /* to be executed at iteration I */
        if(j==1 && 1<=i && 2*i<=n){
          ip = 2*i;  jp = 1;
          add J=[ip,jp] to S_tmp; }
        if(1<=i && i<=n && 1<=j&&j<n){
          ip=i;  jp = 1 + j;
          add J=[ip,jp] to S_tmp; }
      }
    S' = S_tmp;
  }
}
```

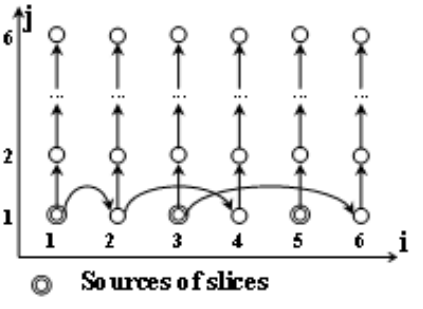

Fig. 2. Iteration space and dependences for the motivating example after applying the algorithm

5 Experiments

We have studied the parallel code above to examine what is its speedup and efficiency. We have carried experiments on a multiprocessor machine (2x Intel

Quad Core, 1.6 GHz, 2 GB RAM, Ubuntu Linux) using the OpenMP API [15]. The time of the execution of the original loop and that of the code produced by our algorithm on one processor as well as speed-up, S, and efficiency, E, for 2,4, and 8 processors are presented in Table 1. The following conclusions can be made analyzing the data in Table 1. There is no considerable difference between the time of the execution of the original sequential *for loop* and that of the *while loop* produced by the presented algorithm. The produced parallel code is characterized by positive speed-up(S>1). The efficiency of the parallel code increases as the volume of computations executed by the produced while loop increases.

Table 1. Results for the motivating example

N	1 CPU		2 CPU		4 CPU		8 CPU	
	for [s]	while [s]	S	E	S	E	S	E
250	0,014	0,016	1,273	0,636	1,167	0,292	1,077	0,135
500	0,056	0,064	1,436	0,718	2,074	0,519	2,667	0,333
1000	0,224	0,257	1,659	0,830	3,027	0,757	4,148	0,519
1500	0,504	0,573	1,732	0,866	3,170	0,792	5,538	0,692
2000	0,922	1,006	1,787	0,893	3,453	0,863	6,188	0,773

We have also studied UTDSP benchmarks [14] to recognize what is the dependence graph topology and what is the effectiveness of the presented algorithm. The UTDSP Benchmark Suite was created to evaluate the quality of code generated by a high-level language (such as C) compiler targeting a programmable digital signal processor (DSP). We have considered only such loops for which Petit [12] was able to carry out a dependence analysis.

Table 2. Results for UTDSP loops

All loops	Topology of dependence graph	MIE graphs		tree		chain	
18	Before using the algorithm	14	78%	1	5%	3	17%
	After using the algorithm	5	27%	2	11%	11	62%

Table 2 presents the number and percentage of loops with different dependence graph topologies. MIE graphs occur in 14 of 18 graphs. After using the presented approach, the number of loops with that topology was reduced to 5. For 14 MIE graphs, the approach is able to transform 9 of them to the tree or chain topology.

We examined parallel codes produced on the basis of the presented approach for UTDSP loops. Experimental results were similar for the loops under examination. Due to the lack of space, we attach experimental results only for the *edge_detect* application from the UTDSP suite. Dependences in this loop are represented by a MIE graph. After applying the proposed algorithm, dependences

Table 3. Experimental results for the UTDSP *edge_detect* application

N	1 CPU		2 CPU		4 CPU		8 CPU	
	for[s]	while [s]	S	E	S	E	S	E
1000	0,047	0,061	1,309	0,654	1,521	0,380	1,842	0,230
1500	0,107	0,137	1,446	0,723	2,099	0,525	2,301	0,288
2000	0,188	0,246	1,440	0,720	2,806	0,701	3,388	0,423
2500	0,294	0,382	1,431	0,716	2,786	0,696	3,406	0,426
3000	0,428	0,550	1,507	0,754	2,828	0,707	4,389	0,549

are represented by chains. The time, speed-up and efficiency of a parallel program (generated by approach [7]) are presented in Table 3. Analysing the data in this table, we can conclude that the presented approach can be applied for producing efficient parallel code for real-life loops.

6 Related Work

The affine transformation framework, considered in papers [4,5,6] unifies a large number of previously proposed loop transformations. However, it fails to extract synchronization-free parallelism when slices are described by non-linear constraints. Our previous papers [7,8] are devoted to the extraction of non-linear synchronization-free slices being represented by graphs of the chain or tree topology only. The techniques presented in those papers fail to extract slices being represented by multiple incoming edges(MIE) graphs.

The main contribution of this paper is the demonstration how to extract synchronization-free slices described by both affine and non-linear constraints and being represented by MIE graphs. The algorithm presented in this paper was implemented by means of the Omega library. Carried out experiments by means of a tool developed [16] demonstrate the correctness and satisfactory efficiency of parallel code produced on the basis of the algorithm presented in this paper.

7 Conclusion and Future Work

We presented a way how to extract synchronization-free slices represented by both affine and non-linear constraints. Code generated by the presented algorithm assigns loop statement instances to be executed according to the free scheduling policy. The algorithm successes when eliminating common dependence destinations does not violate the free scheduling valid for an original dependence graph presenting exact dependences in a program loop. The algorithm can be applied to produce parallel coarse-grained programs for parallel shared memory computers. Our future research direction is to derive techniques for extracting both synchronization-free slices and slices requiring synchronization and being represented by graphs of an arbitrary topology.

References

1. Akhter, S., Roberts, J.: Multi-core Programming, Intel Books by Engineers, for Engineers (2006)
2. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures, p. 790. Morgan Kaufmann, San Francisco (2001)
3. Banerjee, U.: Unimodular transformations of double loops. In: Proceedings of the Third Workshop on Languages and Compilers for Parallel Computing, pp. 192–219 (1990)
4. Feautrier, P.: Toward automatic distribution. Journal of Parallel Processing Letters 4, 233–244 (1994)
5. Lim, W., Cheong, G.I., Lam, M.S.: An affine partitioning algorithm to maximize parallelism and minimize communication. In: Proceedings of the 13th ACM SIGARCH International Conference on Supercomputing (1999)
6. Darte, A., Robert, Y., Vivien, F.: Scheduling and Automatic Parallelization. Birkhuser, Boston (2000)
7. Bielecki, W., Beletska, A., Palkowski, M., San Pietro, P.: Extracting synchronization-free chains of dependent iterations in non-uniform loops. In: 14th International Multi-Conference on Advanced Computer Systems, Polish Journal of Environmental Studies, Miedzyzdroje, vol. 16(5B), pp. 165–171 (2007)
8. Bielecki, W., Beletska, A., Palkowski, M., San Pietro, P.: Extracting synchronization-free trees composed of non-uniform loop operations. In: Bourgeois, A.G., Zheng, S.Q. (eds.) ICA3PP 2008. LNCS, vol. 5022, pp. 185–195. Springer, Heidelberg (2008)
9. Boulet, P., Darte, A., Silber, G., Vivien, F.: Loop paralellelization algorithms: from parallelism extraction to code generation, Technical Report 97-17, LIP, ENS-Lyon, France, p. 30 (June 1997)
10. The Omega project, `http://www.cs.umd.edu/projects/omega`
11. Pugh, W., Wonnacott, D.: An exact method for analysis of value-based array data dependences. In: The Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing. Springer, Heidelberg (1993)
12. Kelly, W., Maslov, V., Pugh, W., Rosser, E., Shpeisman, T., Wonnacott, D.: The omega library interface guide. Technical report, College Park, MD, USA (1995)
13. Pugh, W., Rosser, E.: Iteration space slicing and its application to communication optimization. In: Proceedings of the International Conference on Supercomputing, pp. 221–228 (1997)
14. UTDSP benchmark suite, `http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html`
15. OpenMP API, `http://www.openmp.org`
16. Implementation of the presented approach source code, and documentation, `http://detox.wi.ps.pl/SFS_Project`