ISSN 2277-3061
Volume 15 Number7
International journal of Computers and Technology

Optimal Parallelization Of Loop Structures

Monica-Iuliana ciaca, Loredana Mocean, Alexandru Vancea, Mihai Avornicului
Babeș-Bolyai University,
monica.ciaca@econ.ubbcluj.ro,loredana.mocean@econ.ubbcluj.ro,
alexandru. vancea@cs.ubbcluj.ro,mihai.avorniculului@econ.ubbcluj.ro

ABSTRACT

This paper is intended to be a follow up of the work done by the authors in previous articles. On one hand it is concluded with a theorem that proves to be a definite answer to one very important research direction and on the other hand it is an opening of new research directions in the field of loop structures automatic parallelization.

Keywords

Loop Structures; Parallel Computing; Optimal Parallelization

Academic Discipline And Sub-Disciplines

Mathematics applied in Computer Science

SUBJECT CLASSIFICATION

JEL Classification: C02, C63

SUBJECT CLASSIFICATION

37N40

TYPE (METHOD/APPROACH)

Quasi-Experimental; Literary

Introduction

In some previous articles ([1] [2] [3] [4]) we have presented and analyzed the most important transformations of source code present at the level of loop structures in restructuring compilers. We pointed out the idea that the problem of choosing an optimal sequence of transformations that leads to the most efficient parallel version remains an open question. Current compilers manage to incorporate only a set of heuristic decisions.

On the other hand, before trying to generate an optimal sequence of looping transformations, it seems natural for us to ask if in general, a program always supports an optimal planning scheme for execution.

The answer to this question is negative and we have reached this result due to the original contribution of this paper. In this paper we demonstrate that an optimal scheduling scheme cannot be built in the case of a general program (even when dealing with enough but finite resources), thus solving the open question formulated in [6].

Optimal parallelization problem

Restructuring compilers exploit the inherent parallelism from sequential programs, having as final purpose the efficient execution through the construction of an optimal planning scheme. In this perspective, the algorithms presented in our previous work are particular techniques for obtaining optimal scheduling scheme for the execution of looping structures.

From the theoretical point of view, a question naturally appears: having an arbitrary loop structure and an architecture model that assumes sufficient but finite resources, is it always possible to construct an optimal scheduling scheme?

In this section we define the notion of optimality of a planning execution scheme and we show that in general such scheme cannot be built, showing that there are looping structures requiring infinite resources in order to support optimal planning schemes[5].

Preliminaries

Restructuring compilers aim at efficient exploitation of parallelism available at the level of a program. Parallelization of fine granularity (i.e. at the level of instruction, also called compacting) exploits irregular parallelism at the level of a loop through concurrent execution of operations belonging to the same iteration. Coarser methods (such as *doacross* [7]) allow extracting a higher degree of parallelism by executing concurrently additional instructions or operations belonging to different iterations.

A lot of attention was given to the parallelization of *doacross* loops ([8],[9]). A *doacross* loop type expresses a certain degree of competition, although preventing a loop's iterations to run in parallel. To obtain an optimal execution of programs that do not contain control structures, a polynomial algorithm called linear planning is used (*scheduling list*), which does not require resource constraints. Planning for execution that requires taking into consideration such restrictions is known to be an NP-complete problem [10]. Assuming the existence of sufficient resources, the only

conditioning that will dictate parallel execution planning of the instructions or operations, remains the one which refers to existing data dependencies at the level of the program code. Planning for parallel execution presumes, in most cases, applying transformations to highlight precisely such possibilities. Obviously they must ensure semantic equivalence between the original and the transformed code. Therefore, we define the notion of equivalent transformation of a program code with the purpose of automated parallelization.

Definition 1. Two sequences of source code are called semantically equivalent if one can be obtained from the other by applying a sequence of transformations which preserve the original data dependencies. A detailed analysis of these changes has been made in previous papers.

Aiken and Nicolau in [6] studied the optimal parallelization of looping structures, building a greedy planning algorithm that detects a so called loop cliché (loop pattern).***However their algorithm requires a severe restriction, namely the lack of conditional instructions.***

The paper concludes with the following statement: "remains an open question whether the optimality results of our work can be extended for looping structures with arbitrary control flow".

We are going to show through an analysis of a simple counter example, that in general (understood as the possibility of using inside a loop any instructions that affect flow control), optimal planning for a source code semantically equivalent to a loop is *impossible to be obtained.*

Most studies and analysis on optimality planning suppose that an architectural model has *sufficient but finite resources*, meaning that the architecture can run any program (i.e. resources are sufficient) whose required resources are limited by an arbitrary large enough integer (i.e. *resources are finite however*).

In the following, we note with R the number of available resources. In order to simplify the analysis, we assume without loss of generality that *any execution of an elementary operation requires one clock cycle.*

Intuitively, by the *optimal parallelization of a program* we understand getting the semantic equivalent form of a program for which we achieve at every *t* moment, the planning in parallel of all independent operations available for execution. Such program is also called the *optimal program*. Therefore, parallelization is optimal if we get through transformations an optimal program.

In the following, we define equivalently the concept of optimal program in three more ways.

Definition 2. A program P is called optimal if one of the followings is true:

a). for each operation *w* of *P*, executed at time *t*, there is a dependence chain with length *t* that ends at *w;*

b). for each execution *E* of *P*, *E* is carried out in the shortest time, relative to data dependencies of *P;*

c). the length of any execution *E* (interpreted as a way of executing in the data dependencies graph of *P)* is the length of the longest chain of dependencies of *P*.

Speculative execution method

The appearance of conditional instructions within a loop makes the static planning by compilation to become insufficient for a balanced load of processors. This is due to the large differences that may exist between the execution times of the branches of the decision structure.

In most cases, these tests cannot be evaluated at compile time. Thus, the scheduler does not have the needed information to decide the appropriate computational load for obtaining reasonable efficiency.

That is why, in programs involving massive presence of decision structures, optimality cannot be achieved without the application of so-called *speculative executions*. This involves execution in advance of any action that becomes available for execution according to the dependency restrictions.

In this sense the definition of optimality for executing a program is respected, independently of the decision branch on which the instruction is, so independently of the result of the evaluation after the test condition.

Such an approach ensures that no processor will remain without computational load (only if the data dependencies require it) due to the lack of planning.

Moreover, after the condition evaluation, the desired results are already available, thus contributing significantly to the increase of execution speed.

Obviously, this can be applied in practice only if the resources are enough to take the speculative execution over. In order to solve the general problem of planning, resource limitation itself is not considered a problem, because at least in theory, we can always extend the architecture with new resources (processors, memory) that at the current time have acceptable cost. We define *speculative execution* as follows:

Definition 3. Let S be an instruction control dependent on a *T* test at the source program level. While executing the program, the statement S is *speculatively executed* if it will be scheduled before or concurrently with the test evaluation *T.*

This will mean that, in order to improve global execution, the system will certainly have to perform unnecessary calculations (the results of which will dispense immediately after the test). There will be variations where S is executed, but its result will not contribute in any way to the final result of program execution.

We illustrate the benefits that speculative execution can bring in the body of a cycle that contains conditional statements. Let the following code be given:

```
        for i := 1 to N do
        begin
                if (x > y) then
                        begin
S1:                             z := f(x);
S2:                             x := g(z);
                        end
                          else
S3:                             x := h(x);
S4:             y := E(x);
        end;
```

Fig 1. Sequential looping

It is to be remembered that we assumed for simplicity that the execution of any elementary instruction requires one clock cycle. The sequential execution of these N iterations requires between *3N* (if all iterations are conducted on the false branch) and *4N* (if all iterations are conducted on the true branch) clock cycles.

Applying speculative execution will require just 2 N + 3 clock cycles, which is evident from Figure 2, where we reduced the 4 steps of a sequential iteration into a compact iteration consisting of two steps.

Thus, for large values of N speculative execution method will reduce the execution time of a sequence to half.

In figure 2, planned operations in a single clock cycle are executed in parallel.

Moment	**Scheduled operations**	
	True branch	**F**alse branch
1	test(x>y); z:=$f(x)$;	x_F := $h(x)$;
2	x_T := $g(z)$	y := $E(x_F)$
3	y := $E(x_T)$; z:=$f(x_T)$	
4, 6, ..., 2k	x_T := $g(z)$ test(x>y)	x_F := $h(x_F)$
5, 7, ..., 2k+1	y := $E(x_T)$; z := $f(x_T)$ The content of iterations on *True* branch	y := $E(x_F)$; The content of iterations on *False* branch z := $f(x_F)$

Fig 2. Optimal planning based on the speculative execution of the cycle from Fig. 1.

Bold lines define what we might call the body model execution cycle in which one iteration requires two clock cycles.

Note that through speculative execution $z := f(x_F)$ false branch is calculated in advance (even if you decide not to follow the next iteration true branch, a situation which will result in not needing z).

Together with the test evaluation, $x_T := g(z)$ is calculated in advance. If the test is evaluated to false, we note however, that we will not need all of these values. The optimality is obtained by adding additional cost relative to the resources.

In conclusion, remember that the degree of possible parallelism to be exploited at the level of a program is limited under ideal conditions only by data dependencies, which is the so-called ***inherent parallelism.***

As we have shown we can adopt a model free of hard restrictions because anytime we can expand technical capabilities of the architecture work. Therefore, most architectural models assume *finite but enough resources.*

Optimality execution loops that contain conditional statements.

We approached the issue of speculative execution to highlight that without the application of such techniques cannot be respected the definition of an optimal execution. Even under these conditions, we further show that there are situations where optimal planning cannot be done.

This is due to conditional instructions which are part of a loop body, where one of the branches may hinder speculative execution of activities belonging to other branches.

Being forced to respect the definition of optimal execution, we get at a certain time *t* the necessity for parallel execution of more instructions than the R resources allow. This result leads us to the conclusion that an optimal execution scheme cannot be constructed for the general case of a loop structure.

Intuitively, we note that conditional instructions combined with the restrictions imposed by data dependencies prevent a balanced load of processors in parallel activities to be planned according to the method of speculative execution and which must respect the definition of optimality.

So, we conclude that the R finite resources are insufficient in the general case, for building optimal planning schemes. This, in the sense that *no matter how big we choose the value of R, we can find a program whose optimal execution requires more than R resources.*

This happens because the number of resources required is a function of the time parameter *t*, which is always an unlimited value, even assuming that any program running on our model of the machine will end its execution.

We now address the intuitively noticed formal aspects above and we will discuss them at the level of an adequate example.

Theorem. The problem of building an optimal planning scheme for a loop structure is irresolvable in the general case.

Demonstration. Notice that it is enough to find a looping structure and a particular execution for which an optimal planning scheme cannot be constructed for the theorem to be proved. Remember that we assumed that every step of implementation requires exactly one clock cycle and the definition of optimal scheduling scheme requires that any operation to be executed immediately after its data input becomes available, without any restriction of access to resources.

Let the following example program be given:

```
        i := 0 ; z := c;
        for i := 1 to N do
        begin
                if (z > y) then
S1:                             z := fi(x)
                        else
S2:                             x := h(x);
S3:             y := E(x,z);
        end;
```

Fig 3. Looping structure and a particular execution

where the existing dependencies are $S_2 \rightarrow S_2$ (auto-dependency carried by iterations), $S_2 \rightarrow S_1$, $S_2 \rightarrow S_3$, $S_1 \rightarrow S_3$, $S_3 \rightarrow T$ and $S_1 \rightarrow T$, where *T* is the test condition.

We shall refer to a particular execution, namely that for which the condition evaluates to false for the first *n1* iterations and *true* for the rest *n2* iterations, with $n2 \leq n1$, so $N = n1 + n2$.

Taking into account the existence of dependencies and considering that the R resources of our machine model do not add any restriction of execution, any optimal scheduling scheme will require $n1 + 3$ clock cycles.

This is evident if we look at Figure 3, in which we illustrated optimal planning of operations in every step of execution (clock loop).

It is easy to see the execution cliché for the first *n1* iterations:

for each *p*, $2 \leq p \leq n1$, iteration *p* execute instruction $x_p := h(x_{p-1})$ at the moment *p* and instruction $y_p := E(x_p,z)$ together with test $(z > y_{p-1})$ at the moment $p+1$.

Let us note that in the meantime that we cannot speculatively execute any instruction from the true branch. In fact, this is precisely the main characteristic of the type of conditional sequence that does not support optimal planning scheme, because the first execution of *S1* must wait the final value of *x* provided by the auto dependent *S2* instruction after execution of *n1* iterations of the false branch.

This value will not change further because the next *n2* iterations will go to ***True*** branch.

So, we now have available all the functions f_i and the value of x enabling simultaneous execution of all assignments to z.

To make this possible, i.e. to ensure consistency, it is necessary to not allow simultaneous access to the z variable.

It is applied here the scalar expansion technique [Bacon94] which preserves each z in a separate memory (remember that R represents sufficient resources).

So, for an optimal execution, we must have at the moment $n1$+1, $n2$+2 operations: that $n2$ assignments for z values together with the evaluation of last test of first $n1$ iterations ($z > y_{n1-1}$) and assignment $y_{n1} := E(x_{n1},z)$.

Once we have available the $n2$ values of z, same reason will require to compute all values y of S_3 at the next moment.

We can do this by applying again the scalar expansion, for y values and knowing that we have enough resources.

So, at the moment $n1$ + 2 we must do $n2$+1 operations: the $n2$ assignments of y and the evaluation of ($z > y_{n1}$).

After that, the only remaining operations are those $n2$ tests ($z_p > y_{n1+p}$), which all can be evaluated at the moment $n1$+3, due to the fact that the compared values are all available.

Our analysis concludes that any scheme for the optimal execution of the program code above will require $n1 + 3$ clock cycles. Also, we need:

$n2$ + 2 resources at moment $n1$ + 1

$n2$ + 1 resources at moment $n1$ + 2

$n2$ resources at moment $n1$ + 3

Moment	Scheduled operations		
1	$z > y_0$		$x_1 := h(x_0)$
2		$y_1 := E(x_1, z)$	$x_2 := h(x_1)$
3	$z > y_1$	$y_2 := E(x_2, z)$	$x_3 := h(x_2)$
4	$z > y_2$	$y_3 := E(x_3, z)$	$x_4 := h(x_3)$
. . .	. . .	. . .	. . .
$n2$	. . .	. . .	. . .
. . .	. . .	. . .	. . .
$N1 - 1$	$z > y_{n1-3}$	$y_{n1-2} := E(x_{n1-2}, z)$	$x_{n1-1} := h(x_{n1-2})$
$n1$	$z > y_{n1-2}$	$y_{n1-1} := E(x_{n1-1}, z)$	$x_{n1} := h(x_{n1-1})$; iterația i= n1
$N1 + 1$	$z > y_{n1-1}$	$y_{n1} := E(x_{n1}, z)$	$z_1:=f_1(x_{n1}), z_2:=f_2(x_{n1}),...,$
$N1 + 2$	$z > y_{n1}$	$y_{n1+1} := E(x_{n1}, z_1)$ ...	... $y_{n1+n2} := E(x_{n1}, z_{n2})$
$N1 + 3$	$z > y_{n1+1}$	$z_2 > y_{n1+2}$ . . .	... $z_{n2} > y_{n1+n2}$

Fig 4. Optimal planning scheme for the above cycle.

The problem in this case is that the number of resources required at a time t is a function of t (Resources ($n1$+1) = $n2$+2 = N-$n1$+2).

In the case we have $N >> R$ with $n1$, $n2 >> R$ also, this means that even in the situation of finite and sufficient resources that we considered for our architectural model (which is still the largest working hypothesis we can accept for practical reasons) in general case we have not sufficient resources to plan in an optimal manner the necessary operations.

So, the problem of building a planning scheme for the optimal execution of a loop structure is unsolvable for the example above.

We conclude, therefore, that in the general case we cannot guarantee, for a given program, obtaining the optimal planning scheme of execution, considering ***finite*** resources.

Conclusions

This article represents an original contribution. We showed that for a loop structure, which contains conditional instructions, an optimal planning scheme cannot be constructed. Thus we consider that the open question formulated by [Aiken88] is solved.

We characterized informally a given type of conditional loop, in this way demonstrating the inexistence of an optimal scheme for the general case. As a further development we intend to analyse and formulate the conditions that a loop structure needs to fulfil in order to admit the optimal planning scheme, as an absolutely innovative research direction.

References

[1] Ciaca, M., Mocean L., Vancea A., 2013. *Analisys of Data dependence. Applications in groups of firms,* International Conference on Informatics in Economy, Bucharest 2013, pp. 202-209, ISBN 2284-7472

[2] Ciaca, M., Mocean L., Vancea A., 2014. Loop Restruturing Transformations for Automatic Programs Paralelization, The 13th International Conference on Informatics in Economy, Bucharest University of Economic Studies Press, pg8-14

[3] Mocean, L., Ciaca M., Vancea A, 2013. *Possibilities of parallel processing at the level of economic programs in groups of firms,* International Conference on Informatics in Economy, Bucharest 2013, pp. 31-36, ISBN 2284-7472

[4] Mocean, L., Ciaca M., Vancea A., 2014. Substitution transformations of loop iterations, The 13th International Conference on Informatics in Economy, Bucharest University of Economic Studies Press, pg.94-97

[5] Vancea, A. 1997. Loop scheduling optimality for parallel execution, in Studia Informatica, vol.XLII, no.2, pp.25 - 32.

[6] Aiken, A. and A.Nicolau, 1988. *Optimal loop parallelization*, in *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, Georgia, June 1988, SIGPLAN Notices 23, 7, July 1988, pp.308-317.

[7] Cytron,R.1986. Doacross: Beyond vectorization for multiprocessors, in Proceedings of the 1985 International Conference on Parallel Processing, Penn State University, August 1986, pp.836-844.

[8] Padua, D. and Wolfe,M. 1986, Advanced compiler optimizations for supercomputers, in *Communications of ACM*, 29, 1986, pp.1184-1201.

[9]H.M.Su and P.C Yew - *Efficient Doacross Execution on Distributed Shared-Memory Multiprocessors*, in Proceedings of the ACM Conference on Supercomputing, November 18-22, 1991, Albuquerque, New Mexico, pp.842-853.

[10] Garey,M. and Johnson,D. 1979. Computers and Intractability , A guide to the theory of NP-completeness, Freeman, New York, 1979.

[11] Brunet,S., Andronache,V. and Nelson, L. Passos Ranette Halverson , 2009. Performance Evaluation of Parallel Implementation of Nested Loop Control Structures

[12]Darte, A. Parallelism Detection in Nested Loops, Optimal Encyclopedia of Parallel Computing, 2011, pp 1429-1442

[13]Weijia Sang et al., 1991. On loop transformations for generalized cycle shrinking, 1991 International Conference on Parallel Processing

[14] Adam, T., K.Chandy and J.Dickson, 1974. A comparison of list schedules for parallel processing systems, in *Communications of the ACM*, 17, pp.685-696.

Author' biography

Monica Iuliana CIACA obtained her bachelor's degree at Babes Bolyai University Cluj-Napoca, in the field of Computer Science. After graduation, she has worked as programmer at the Institute for Computation Techniques from Cluj-Napoca. In 1994 she started working at the Babes Bolyai University as teaching assistant, being interested in artificial intelligence, expert systems, business information systems and software engineering. She published various articles, the most important being the one written after her participation in a Tempus Phare project, in Perugia. In 2003 she got her PhD in Mathematics and Computer Science, with a thesis on parallel computing: "Implementation Techniques in Parallel Computing". In the last five years she looked to extend her knowledge in another field: theology. She obtained her Bachelor's Degree and Master's Degree in Biblical Studies and Iconographic exegesis, in 2012, at Babes Bolyai University. Since 2004 she is Associate Professor at Babes Bolyai University, Cluj-Napoca, Faculty of Economics, in the Department of Business Information Systems.

Loredana MOCEAN has graduated Babes-Bolyai University of Cluj-Napoca, the Faculty of Computer Science, she holds a PhD diploma in Economics and she had gone through didactic position of assistant, lecturer and associate professor, since 2000 when she joined the staff of the Babes- Bolyai University of Cluj-Napoca, Faculty of Economics and Business Administration. Also, she graduated Faculty of Economics and Business Administration. She is the author of more than 20 books and over 35 journal articles in the field of Databases, Data mining, Web Ser-vices, Web Ontology, ERP Systems and much more. She is member in more than 20 grants and research projects, national and international.

Alexandru VANCEA has graduated the Computer Science Department of “Babes-Bolyai” University Cluj-Napoca in 1986. Ph.D. in Computer Science in 2000. Research areas and domains of interests: Programming Languages Design and Analysis, Automatic parallelization of programs, Distributed Programming. Teaching: Operating Systems, Computer Architecture, Fundamentals of Programming Languages.

Mihai-Constantin AVORNICULUI has graduated Faculty of Mathematics and Computer Science, Babes-Bolyai University Cluj-Napoca in 2004. He has finished his PhD studies in 2009. He works at the Business Information Systems department of FSEGA, Babes-Bolyai University Cluj-Napoca. His research interests include data mining, information systems, and advantage database systems.