STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **XLVIII**, Number 1, 2003

SCHEDULING OPTIMALITY FOR THE PARALLEL EXECUTION OF LOGIC PROGRAMS

MONICA VANCEA AND ALEXANDRU VANCEA

Abstract. Logic programming is the most widespread programming paradigm used in artificial intelligence, a domain which needs intensive computing resources. Parallel execution of logic programs is the most effective speedup factor which can be applied for obtaining reasonable execution time in some cases. Parallelizing compilers have the task to exploit the inherent parallelism from the sequential programs having as the ultimate goal their efficient execution by means of building a time optimal schedule. These tools focus on the inherent parallelism available at the level of the logic languages operational semantics. Besides particular techniques for achieving optimal execution for specific classes of logic programs, one question arises naturally: given an arbitrary logic program and a machine model which assumes sufficient but finite resources, is it always possible to build a time optimal schedule ? This paper defines the notion of time optimality and proves that in the general case, no time optimal schedule can be built for a logic program, because there are classes of logic programs which require infinite resources for accepting time optimal schedules.

1. Preliminaries

Logic programming is the most popular programming model used in the area of artificial intelligence. When using huge knowledge based systems, acceptable run time for receiving the output results can be obtained only by implementing parallel execution among the tasks generated by the corresponding goals. So, parallel logic programming is sometimes the only possibility to deal with the complexity of some artificial intelligence problems and on the other hand can be an adequate solution for many significant run time optimizations even for the actual solutions [2, 4, 6, 10].

Parallelizing compilers try to automatically and efficiently exploit the parallelism available in a given program, transforming the programs into their parallel

Received by the editors: January 15, 2003.

2000 *Mathematics Subject Classification.* 68M20, 68N17, 68Q10.

1998 *CR Categories and Descriptors.* D.1.6. [**Software**]: Programming Techniques – *Logic Programming*; D.1.3. [**Software**]: Programming Techniques – *Parallel Programming*; D.2.9. [**Software**]: Software Engineering – *Cost Estimation*; D.4.1. [**Software**]: Operating Systems – *Scheduling*.

versions taking into account the existing data dependences [5, 7]. Naturally, this transformations must assure that the semantics are the same.

Definition. Two program codes are said to be ***semantic equivalent*** if one of them can be obtained from the other by applying a sequence of data dependence preserving transformations.

The vast majority of studies upon optimality assume that the machine model has ***sufficient but limited resources***, meaning that the architecture can run any program for which the number of resources needed is bounded by some arbitrary integer [1, 8, 9]. We will denote it further as R and we make some standard assumptions about the operations to be scheduled: for simplicity, we assume, without loss of generality, that *any resolution step of an elementary clause takes one machine cycle.*

Informally, through optimal parallelization of a program code, we understand obtaining a semantically equivalent (SE) version of it which manages at every moment t to schedule in parallel the execution of all its independent operations. That's why we will characterize such a program code as *time optimal*. Thus, the parallelization process is optimal if we obtain a SE time optimal program code.

Formally, we give below three alternative definitions of this concept.

Definition. A program code P is said to be ***time optimal*** if any of the following statements is true:

a) for every operation w executed at moment t, there exists a dependence chain of length t which ends at w;
b) every execution E of P is running in the shortest possible time with regard to the P's data dependences;
c) the length of any execution E (interpreted as an execution path in the data dependence graph of P) is the length of the longest data dependence chain from P.

2. Forward execution

When dealing with conditional predicates at the level of some loopings (recursive calls for example), static scheduling alone cannot assure processors workload balance, due to possible strongly different execution times required by the different branches of such a decision structure. In general, these tests can not be evaluated at compile time, so a scheduler has no information on which can decide a proper load balance for obtaining a reasonable efficiency.

That is why in branch intensive programs time optimality cannot be achieved without *forward execution* of branches, that is executing everything it can be executed (as the time optimality definition requires) on every branch in advance, independently of the results of decision testing. This assures that no processor will be idle and that after evaluating the decision, the results will already be there, computed, thus contributing to a significant speedup.

```
predicates
  p(integer,integer,integer).
  q(integer,integer,integer,integer).
clauses
  p(X_init,Y_init,0) :- !. % starting values for the iterative loop
  p(X,Y,N) :- N1 is N-1, p(X1,Y1,N1), q(X,Y,X1,Y1).
  q(X,Y,X1,Y1) :- X1>Y1, Z is f(X1), X is g(Z), Y is e(X). (A)
  q(X,Y,X1,Y1) :- X1<=Y1, X is h(X1), Y is e(X). (B)
```

FIGURE 1. Code sequence illustrating the use of forward execution

We can define forward execution as follows:

Definition. Let S be a clause which is resolution dependent on a test clause T in a logic program code. During the execution of the program code, the clause S is said to be ***forward executed*** if it will be scheduled before or concurrent with the resolution of T.

This means that the system will do useless work for obtaining better results. Thus, there will be execution histories for which S will execute but its result will not contribute to the program's final output in any way.

We illustrate below the potential benefits of applying forward execution for the component clauses of a looping (recursive) predicate which has conditional clauses. Let's consider the code sequence from Figure 1, which illustrates a conditional iterative process (the input-output flow patterns are $p(o, o, i)$ and $q(o, o, i, i)$ and f, g, h and e are numeric functions).

The sequential execution of the N calls requires between $3N$ (if all calls will choose the B branch) and $4N$ (if all calls will choose the A branch) machine cycles (remember that we assumed for simplicity that every elementary clause resolution takes one cycle). With forward execution $2N + 3$ cycles are needed (evident from table 1, where we reduced the 4 sequential iteration steps to a compact 2 resolution steps iteration), so for large N values the method will improve the runtime execution by a factor of 2. Obviously, the operations scheduled in a time step are executed in parallel. In Table 1 we illustrate what we can call the execution model of the looping predicate, in which a call requires two units of time.

Let's notice that, by forward execution, we compute $z := f(x_B)$ in advance on the false branch (variant B, even if maybe the next iteration won't take the true branch path so no z value will be needed) and together with the test evaluation we also compute in advance $x_A := g(z)$. However, if the test output is false we do not need this values at all. So, time optimality is achieved by adding an extra resources cost.

Time step	**Scheduled operations**			
	True branch		**False branch**	
1	test$(x > y)$; $z := f(x)$; $x_B := h(x)$;			
2	$x_A := g(z)$		$y := e(x_B)$	
3	$y := e(x_A)$; $z := f(x_A)$			
4, 6, ..., $2k$	$x_A := g(z)$	test$(x > y)$		$x_B := h(x_B)$
5, 7, ..., $2k+1$	$y := e(x_A)$; $z := f(x_A)$ Recursive call on the (A) branch (True)		$y := e(x_B)$; Recursive call on the (B) branch (False)	$z := f(x_B)$

TABLE 1. Time optimal schedule with forward execution for the code in Figure 1

Ideally, making abstraction of the real execution conditions, which may vary a lot from case to case, the amount of parallelism which can be exploited is limited only by the data dependences between the program's statements (this defines the so called ***inherent parallelism***). Hardware resource constraints, such as processors and memory, may be eliminated, at least in theory, because we always may add extra technical components to overcome the lack of resources. That is why the most widespread execution models assume as we mentioned *finite but unlimited resources.*

3. Time optimality for logic programs containing conditional clauses

We approached forward execution in section 2 because it is evident that for obtaining a time optimal schedule we have to apply it. Anyway, we will show that there are cases when a time optimal schedule cannot be built with finite resources. This can happen because of the conditionals which are part of the body of a recursive predicate, when one branch can prevent the forward execution of some activities belonging to the other branch. Then, it can be the case that time optimal scheduling (based on its definition) forces at some moment t the parallel execution of much more statements than the R resources can afford so we will conclude that no time optimal schedule can be built for general logic programs. Intuitively, we observe that conditional predicates combined with data dependence restrictions prevent the load balancing of the activities which have to be scheduled in parallel, by means of forward execution and obbeying the definition of time optimality. This makes the finite but unlimited R resources of our machine model to be insufficient for building a time optimal schedule in the general case. This is because the number of resources needed becomes a function of time t, an unbounded value, even if we accept that any program run on our machine model will eventually finish its execution. So, *general practical optimal scheduling is an intractable problem.*

```
predicates
  p(integer,integer,integer).
  q(integer,integer,integer,integer,integer,integer,integer).
clauses
  p(X_init,Y_init,Z_init,0) :- !. % start values for iterative loop
  p(X,Y,Z,N) :- N1 is N-1, p(X1,Y1,Z1,N1), q(X,Y,Z,X1,Y1,Z1,N1).
  q(X,Y,Z,X1,Y1,Z1,N1) :- Z1>Y1, Z is f(X1,N1), Y is e(X1,Z). (A)
  q(X,Y,Z,X1,Y1,Z1,N1) :- Z1<=Y1, X is h(X1), Y is e(X,Z1). (B)
```

FIGURE 2. Code sequence illustrating a conditional iterative process

We will ellaborate more formally on this intuitive observations in the following, taking a suitable example to illustrate our point of view.

Theorem 1. *A general logic program has no time optimal schedule.*

Proof. Let's notice that if we find only one class of logic programs and only one particular execution history for which no time optimal schedule can be built then our result holds. Recall that we assumed that any resolution step requires exactly one time unit and that the definition of a time optimal schedule requires any operation to be executed immediately after its input data become available (without any supplimentary resource access restriction). So, we can consider for example the program code in Figure 2.

There, the input-output flow patterns are $p(o,o,o,i)$ and $q(o,o,o,i,i,i,i)$. Predicate p is a iterative recursive predicate which associates at any iteration current values for the variables X, Y and Z by calling the q predicate. Current X, Y and Z values are computed using the X, Y and Z values from the previous iteration (these previous values are identified in the above code by variables $X1$, $Y1$ and $Z1$ and they are passed as parameters from one iteration to the next by means of the recursive call of the p predicate). We will identify some particular predicates from the above code as follows:

S_0 $\equiv$ p(X,Y,Z,N); $S_1 \equiv$ Z is f(X1,N1); S_2 $\equiv$ X is h(X1);
S_3 $\equiv$ Y is e(X1,Z); $S_4 \equiv$ Y is e(X,Z1); T_1 $\equiv$ Z1>Y1; $T_2 \equiv$ Z1<=Y1;

Using these notations we can identify the following dependences:

S_0 $\rightarrow$ S_0 (recursive self-dependence for accomplishing the iterative process);

S_2 $\rightarrow$ S_4 (output X from S_2 is used as input in S_4);

S_1 $\rightarrow$ S_3 (output Z from S_1 is used as input in S_3);

S_2 $\rightarrow$ S_1 (X1 used as input in S_1 is in fact the output X from the S_2 previous iteration);

S_2 $\rightarrow$ S_3 (X1 used as input in S_3 is in fact the output X from the S_2 previous iteration);

$S_2 \rightarrow S_2$ (iteration carried self-dependence – X1 used as an input parameter for function h is in fact the output X from the previous iteration of the same clause);

$S_1 \rightarrow T_1$,T_2 (the output Z from S_1 is used as input in the tests T_1 and T_2);

S_3, $S_4 \rightarrow T_1$,T_2 (the output Y from S_3 or S_4 is used as input in the tests T_1 and T_2);

We will refer further to a particular execution history, namely the one that takes the (A) branch for the first n1 iterations (T_1 evaluated to *false* and T_2 evaluated to *true*) and the (B) branch for the remaining n2 ones (T_2 evaluated to *false* and T_1 evaluated to *true*), with n2 $\leqslant$ n1. So, $N = n1 + n2$. Taking into account the identified dependences and considering that the R resources of our machine model do not add any other execution restrictions, any time optimal execution of our program code will need $n1 + 3$ time units (remember that we assume that every execution step requires exactly one machine cycle – or time unit – and that the definition of a time optimal schedule requires that any operation takes place as soon as the inputs are available, with no resource constraints). This becomes evident looking at Table 2 where we show which operations are executing at each time step. It is easy to see the pattern for the first $n1$ iterations (recursive calls): for each k, $2 \leqslant k \leqslant n1$, iteration k executes the clause $x_k := h(x_{k-1})$ (S_2) at time step k and the clause $y_k := E(x_k, z)$ (S_4) together with the test T_2 at the time step $k + 1$.

We must notice that in the meantime no statement is available for the forward execution on the (A) branch, due to the fact that the first execution of S_1 must wait on the final value of X issued by the self dependent predicate S_2 after the n1 iterations on the (B) branch. This value for X will not change further at all (we said that the next $n2$ iterations all will go on the (A) branch). So, we have now the function f and the last value of X available (from the $n1$ time step), making theoretically possible that all the bindings to the Z variables to be made simultaneously (in the same time unit). This can be done if we apply scalar expansion [3] for being able to retain every instance of Z in a separate memory cell (remember that we have finite but sufficient resources). So, for a time optimal execution we must have at the time step $n1+1$, $n2+2$ operations: the $n2$ bindings for Z's together with the last test of the first $n1$ iterations (T_2) and the binding $y_{n1} := E(x_{n1}, z)$. After we have all the Z values, the same reasons force us to compute all the data dependent Y values of S_3 in the next time step. We can do this applying again scalar expansion for the Y values and knowing that we have enough resources for this. So, for a time optimal execution we must have at the time step $n1 + 2$, $n2 + 1$ operations: the $n2$ bindings for the Y instances and the evaluation of the test T_1. After that, the only remaining operations are the $n2$ tests T_1 which all can be evaluated in the time step $n1 + 3$, because the values being compared are now all available.

Time step	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	$z > y_2$	$y_3 := E(x_3, z)$	$x_4 := h(x_3)$
. . .	. . .	. . .	. . .
$n2$	. . .	. . .	. . .
. . .	. . .	. . .	. . .
$n1 - 1$	$z > y_{n1-3}$	$y_{n1-2} := E(x_{n1-2}, z)$	$x_{n1-1} := h(x_{n1-2})$
$n1$	$z > y_{n1-2}$	$y_{n1-1} := E(x_{n1-1}, z)$	$x_{n1} := h(x_{n1-1})$; *iteration* $i = n1$
$n1 + 1$	$z > y_{n1-1}$	$y_{n1} := E(x_{n1}, z)$	$z_1 := f_1(x_{n1})$, $z_2 := f_2(x_{n1})$, $\ldots$, $z_{n2} := f_{n2}(x_{n1})$
$n1 + 2$	$z > y_{n1}$	$y_{n1+1} := E(x_{n1}, z_1) \ldots$	$\ldots y_{n1+n2} := E(x_{n1}, z_{n2})$
$n1 + 3$	$z > y_{n1+1}$	$z_2 > y_{n1+2} \cdots$	$\ldots z_{n2} > y_{n1+n2}$

TABLE 2. Time optimal schedule for the code sequence in Figure 2

So, our analysis reveals that any time optimal execution of the above program code will need $n1 + 3$ time units. Also, we need

- $n2 + 2$ resources in the $n1 + 1$ time step;
- $n2 + 1$ resources in the $n1 + 2$ time step;
- $n2$ resources in the $n1 + 3$ time step.

The problem in this case is that the number of resources (processors) needed at a time step is a function of that time step (Resources$(n1+1) = n2+2 = N-n1+2$). But if we have $N >> R$ with $n1$, $n2 >> R$ also, this means that ***even with the finite but sufficient resources*** that we considered in our machine model (the largest assumption we can made anyway for practical purposes) ***we have not enough resources available to schedule the required operations for an optimal execution***. So, no optimal schedule exists for the execution of the above logic program code.

We conclude then, that in the general case, no time optimal schedule is guaranteed to be found for a given program considering ***finite*** resources. ■

4. Conclusions and future work

We showed in section 3 that a time optimal schedule cannot be built with finite resources for a general logic program. We informally characterized one class of logic programs (namely those containing conditional predicates) that has no time optimal schedule and explained why it is so. This was sufficient for proving

that no time optimal schedule exists for a logic program in the general case. A thorough analysis of logic program features which determine the conditions under time optimal schedules exists is what we intend to do as future research.

References

[1] **F.E.Allen**, ***Program optimization***, in *Annual Review in Automatic Programming 5, International Tracts in Computer Science and Technology and their Applications*, vol.13, Pergamon Press, Oxford, England, 1969, 239–307.

[2] **K.A.M. Ali, R. Karlsson**, ***OR-Parallel Speedups in a Knowledge Based System: on Muse and Aurora,*** in *FGCS'92*, Tokyo, 1992.

[3] **D.Bacon, S.Graham and O.Sharp**, ***Compiler Transformations for High-Performance Computing,*** in *ACM Computing Surveys*, vol.26, no.4, December 1994, 345–420.

[4] **R. Bahgat and S. Gregory**, ***Pandora: Non-deterministic Parallel Logic Programming***, in G.Levi and M.Martelli, editors, *Proc. of the 6th International Conference on Logic Porgramming*. MIT Press, 1990.

[5] **Utpal Banerjee**, ***Dependence Analysis***, Kluwer Academic Publishers, 1997.

[6] **K.L. Clark, S. Gregory**, ***PARLOG: Parallel Programming in Logic***, in *A.C.M. TOPLAS*, Vol. 8, No.1, Jan. 1986.

[7] **Grigor Moldovan, Alexandru Vancea, Monica Vancea**, ***Data dependence testing for automatic parallelization***, Studia Univ. Babeş-Bolyai, Informatica, vol.II, no.1, 1997, 3–18.

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

[9] **Alexandru Vancea and Monica Vancea**, ***Efficient Parallel Code Generation for nested for-loops*****,** Seminar on Computer Science, Preprint no.2, 1997, 179–188.

[10] **Monica Vancea**, ***Executia paralela a programelor logice***, Sesiunea de Comunicări Ştiinţifice *Economia României la orizontul anului 2000*, 14-15 noiembrie 1998, Cluj-Napoca, in *Studii şi Cercetări Economice*, vol. XXVIII–XXIX, 1988, 985–995.

Faculty of Economic Science, Babeş-Bolyai University, Cluj-Napoca, Romania
E-mail address: vancea@econ.ubbcluj.ro

Faculty of Mathematics and Computer Science, Babeş-Bolyai University, Cluj-Napoca, Romania
E-mail address: vancea@cs.ubbcluj.ro