STUDIA UNIV. "BABEŞ–BOLYAI", INFORMATICA, Volume **XLII**, Number 2, 1997

Loop Scheduling Optimality for Parallel Execution

ALEXANDRU VANCEA

Abstract. 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 concentrate on the parallelism available in loops, while a program is spending there most of its execution time. Besides particular techniques for achieving optimal execution for specific loops, one question arise naturally: given an arbitrary loop 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 loop time optimal schedule can be built, because there are loops which require infinite resources for accepting time optimal schedules.

1. Introduction

Parallelizing compilers try to efficiently exploit the parallelism available in a given program, particularly parallelism that is too low-level or irregular to be expressed by common coding. Fine grain (instruction level) parallelization, called *compaction*, captures irregular parallelism inside a loop body by concurrently executing several operations of the same iteration. Coarser methods, such as ***doacross*** [Cytron86] allow the extraction of much more parallelism by concurrently executing statements or operations belonging to distinct iterations.

Much attention has been devoted to the parallelization of *doacross* loops [Padua86, Munshi87, Su91]. A doacross loop expresses some recurence, preventing the iterations of the loop from executing independently.

A polynomial time algorithm, named *list scheduling* [Adam74] is usually applied for obtaining a time optimal solution for acyclic straight line programs in the case of unlimited resources. Scheduling with resource constraints is known to be NP-hard [Garey79].

Received by the editors: February 7, 1998.

1991 *Mathematics Subject Classification.* 68N15.

1991 *CR Categories and Descriptors.* D.1.3 [**Programming Techniques**]: Concurrent Programming – *parallel programming*; D.2.8 [**Software Engineering**]: Metrics – *performance measures*; D.3.4 [**Programming Languages**]: Processors – *optimization, compilers.*

Parallelizing compilers transform the programs into their parallel versions taking into account the existing data dependences. Naturally, this transformations must assure that the semantics are the same.

Definition 1.1. *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.*

A survey of the most widely used such transformations can be found in [Padua86].

Aiken and Nicolau [AN88] studied optimal loop parallelization, yielding an optimal greedy scheduling algorithm which detects a *loop pattern*. Their major restriction is that the loop body should contain no if statements, but only straight line code. Their research ends by stating that "it is an open problem whether the optimality results of this paper can be extended to loops with arbitrary flow of control". We will show in this paper by a simple analysis of a counter example, that in the general case (understood as using any kind of control flow statements), finding a time optimal schedule for a semantic equivalent code is not possible.

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. We will denote it further as R and we make some standard assumptions about the loops to be scheduled: for simplicity, we assume, without loss of generality, that *any operation 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 1.2. *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.*

```
    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
```

FIGURE 1. A sequential loop

2. Forward execution

When considering **if** control statements, 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.

We can define forward execution as follows:

Definition 2.1. *Let S be a statement which is control dependent on a decision test T in a program code. During the execution of the program code, S is said to be* ***forward executed*** *if it will be scheduled before or concurrent with 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 a loop's statements. Let's consider the following loop:

The sequential execution of the loop requires $4N$ machine cycles for the N iterations (remember that we assumed for simplicity that every sequential statement - or execution step - takes one cycle). With forward execution $2N + 3$ cycles are needed (evident from table 1, where we reduced the 4 steps iteration to a compact 2 steps iteration), so for large N the method will improve the runtime execution

Time step	Scheduled operations	
	True branch	False 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	$y := E(x_T)$; $z := f(x_T)$ One loop iteration on the True branch	$y := E(x_F)$; One loop iteration on the False branch / $z := f(x_F)$ loop body execution pattern (one iteration takes 2 time steps)

TABLE 1. Time optimal schedule with forward execution for the loop from figure 1.

by a factor of 2. Obviously, the operations scheduled in a time step are executed in parallel.

Let's notice that, by forward execution, we compute $z := f(x_F)$ in advance on the false branch (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 $x_T := g(z)$. 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.

Ideally, making abstraction of the real execution conditions, which may vary a lot from case to case, the amount of loop parallelism which can be exploited is limited only by the data dependences between the loop's statements or iterations (this forms 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 components. That is why the most widespread execution models assumes as we mentioned *finite but unlimited resources*.

3. Time optimality for loops with conditional statements

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 a loop, when one branch can prevent the forward execution of some activities of 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 conditional loops. Intuitively, we observe that conditionals 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.*

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

Theorem 3.1. *A general control flow loop has no time optimal schedule*

Proof. Let's notice that if we find only one loop and only one particular execution history for which no time optimal schedule can be built then our result holds. So, we can consider for example the following program code:

$i := 0$; $z := c$;
    for $i := 1$ **to** N **do**
    begin
        if $(z > y)$ **then**
S_1: $z := f_i(x)$
        else
S_2: $x := h(x)$
S_3: $y := E(x, z)$
end;

which has the dependences $S_2 \rightarrow S_2$ (loop carried self dependence), $S_2 \rightarrow S_1$, $S_2 \rightarrow S_3$, $S_1 \rightarrow S_3$, $S_3 \rightarrow T$ and $S_1 \rightarrow T$ where T is the loop's test.

We refer further to a particular execution history, namely the one that takes the false branch for the first $n1$ iterations and the true branch for the remaining $n2$ ones, with $n2 = n1$. So, $N = n1 + n2$. Taking into account the 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 $n1 + 3$ machine cycles (remember that we assume that every execution step requires exactly one machine cycle 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: for each p, $2 \leq p \leq n1$, iteration p executes the statement $x_p := h(x_{p-1})$ at cycle p and the statement $y_p := E(x_p, z)$ together with the test $(z > y_{p-1})$ at cycle $p + 1$.

We must notice that in the meantime no statement is available for the forward execution on the true 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 statement S_2 after the $n1$ iterations on the false branch. This value will not change further at all (we said that the next n2 iterations all will go on the true branch). So, we have now available all the functions f_i and the value x, making possible in theory that all the assignments to z to be made simultaneously (in the same machine cycle). This can be done if we apply scalar expansion [Bacon94] for being able to retain every 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 $n2$ assignments to z's together with the last test of the first n1 iterations ($z > y_{n1-1}$) and the assignment $y_{n1} := E(x_{n1}, z)$. 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 $n2$ assignments to y's and the evaluation of the test ($z > y_{n1}$). After that, the only remaining operations are the $n2$ tests ($z_p > y_{n1+p}$) which all can be evaluated in the time step $n1+3$, because the values being compared are now all available.

So, our analysis reveals that any time optimal execution of the above program code will need n1 + 3 machine cycles. 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 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 considered 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 (although *unlimited*). □

4. Conclusions and future work

We showed in section 3 that a time optimal schedule cannot be built with finite resources for a loop with general control flow. So, we addressed and solved the open problem posed in [AN88]. We informally characterized one kind of conditional loop that has no optimal schedule and explained why it is so. This was sufficient for proving that no optimal schedule exists for a *conditional* loop in the general case.

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	$z > y_{n1}$	$y_{n1+1} := E(x_{n1}, z_1)$ ...	... $y_{n1+n2} := E(x_{n1}, z_{n2})$
n1 + 3	$z > y_{n1+1}$	$z_2 > y_{n1+2} \ldots$	... $z_{n2} > y_{n1+n2}$

TABLE 2. Time optimal schedule for the above loop.

This implies that no optimal schedule exists for a loop in the general case. A thorough analysis of loop features which determine the conditions under optimal schedules exists is what we intend to do further.

References

[Adam74] T. Adam, K. Chandy and J.Dickson, *A comparison of list schedules for parallel processing systems*, Comm. of the ACM, 17, 1974, pp. 685-696.

[Bacon94] D. Bacon, S. Graham and O. Sharp, *Compiler Transformations for High-Performance Computing*, ACM Computing Surveys, vol.26, no.4, December 1994, pp. 345-420.

[AN88] A. Aiken and A. Nicolau, *Optimal loop parallelization* Proceedings of the 1988 SIGPLAN Conference on Programming Language Design and Implementation, Atlanta, Georgia, June 1988, pp. 221-235.

[Cytron86] R. Cytron, *Doacross: Beyond vectorization for multiprocessors* Proceedings of the 1985 International Conference on Parallel Processing, Penn State University, August 1986, pp.836-844.

[Garey79] M. Garey and D. Johnson, *Computers and Intractability - A guide to the theory of NP-completeness* Freeman, New York, 1979.

[Munshi87] A. Munshi and B. Simmons, *Scheduling sequential loops on parallel processors*, Technical Report 5546, IBM, 1987.

[Padua86] D. Padua and M. Wolfe, *Advanced compiler optimizations for supercomputers* Comm. of ACM, 29, 1986, pp. 1184-1201.

[Su91] 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.

"BABEŞ-BOLYAI" UNIVERSITY, FACULTY OF MATHEMATICS AND INFORMATICS, RO-3400 CLUJ-NAPOCA, ROMANIA

E-mail address: vancea@cs.ubbcluj.ro