A Note on Scheduling Algorithms for Processors with Lookahead

Cristian Ungureanu
New York University

June 1996

1 Introduction

Many superscalar processors designed today are able to dynamically schedule instructions. Dynamic scheduling means that a processor is able to analyze a portion of the instruction stream "on the fly", and has the capability of issuing an instruction other than the next one available in the input, in order to avoid stalling. Such an instruction is said to be executed *out of order*.

Scheduling algorithms for machines with in-order execution are used in most compilers today. However, schedules which are optimal for machines with in-order execution may be sub-optimal for a machine with out-of-order execution. Optimization algorithms to take advantage of dynamic scheduling are still a subject of research. After we make our model precise, we will describe an algorithm which produces a *local schedule* for a trace of basic blocks such that the *completion time* is minimized for a processor with a depth of the pipeline $k = 2$ and dynamic scheduling ability. The algorithm runs in polynomial time for any fixed value of the *scope* size, s. However, the larger s is, the higher the complexity of the algorithm. Our intuition tells us that the algorithm is NP-hard if s is part of the input, but we have no proof yet.

2 Program representation

The input to our algorithm is an instruction trace. *Traces* are loop-free linear sections of code, which can span several basic blocks. A trace is formally represented as a weighted *D*irected *A*cyclic *G*raph (or *DAG*) $G = (V, E, BB, w)$, where each instruction i on the trace is represented as a node in V. We use $BB(i)$ to denote the basic block in which i occurs in the program. Since no ambiguity can arise, we'll also denote by $BB(i)$ the position of the basic block in the trace; i.e. BB is also a function which maps basic blocks into the set $\{1,\ldots,L\}$ (where L is the number of basic blocks), such that $BB(x) < BB(y)$ if control flows from basic block x to basic block y.

The edges E in the trace represent data dependences. There is a directed edge from node i to node j whenever instruction i has to be executed before instruction j.

The instruction stream is executed on a pipelined machine and hence, might involve additional delays during execution due to latencies. Specifically, given an instruction i issued at time $\tau(i)$, it may not always be possible to issue another instruction j at a later time $\tau(i) + t, t > 0$, because the result(s) of i are not yet available and j uses them. This delay is referred to as the *inter-instruction latency* (or *latency* for short) of i relative to j, and is modeled by the integer weight $w(i, j)$ given to edge (i, j). In this paper, the only possible values for the latency are 0 or 1 (corresponding to a functional unit with a pipeline of depth two).

The output of the algorithm is a schedule, which for our purpose is just a linear sequence of instructions[1] which contains an idle slot between any two consecutive instructions which are linked by an edge with latency equal to 1. The position of the instructions in the list imposes on them a total order, which must be consistent with the partial order given by the data dependency. If it is also consistent with the order of the basic blocks, the schedule is called *local*; otherwise, it is called *global*.

3 The Processor

In our model, the processor is able at any given cycle to inspect a portion of the instruction stream, called *scope* which may consist of one or more instructions. The number of instructions in scope s, is a fixed parameter of the architecture. By analyzing[2] the dependences that instructions in scope have on previously issued instructions (and also among themselves), the processor can decide which of them are *ready* (i.e. can be issued). According to some policy, the processor then chooses one of the ready instructions to be issued in the current cycle. In this paper we make the assumption that the policy followed by the processor is to issue the "oldest" ready instructions available in scope (i.e. it uses a first-in first-out order). If no ready instruction is available in scope, the processor stalls one time step. After the processor issues an instruction i, the instruction is removed from the scope, and the next instruction from the stream is enqueued to the scope. The number of time steps it takes the processor to issue all instructions from the stream is called *completion time*.

It is important to note that this model is just one of the many possible for a processor with out-of-order execution. Although our processor uses a FIFO order, it is also possible to dynamically rank the instructions inside the scope according to some heuristics, and issue them according to their ranks. Also, the processors may differ in regards to what happens to instructions issued out-of-order. Our model requires them to be excluded from the scope. However,

[1] We are scheduling traces for a machine with only one functional unit.

[2] The processors have special hardware support to analyze the current program dependences "on-the-fly".

it is also possible that such instruction remain in scope until all instructions which precede it have been issued. These differences in processors may lead to a different execution order of instructions. Consequently, the algorithm we describe will not necessarily produce an optimum schedule for such a machine. For a more detailed discussion of the possible hardware models for out-of-order execution, and scheduling algorithms to take advantage of it, see [LPU95].

4 Scheduling Algorithm

The goal of instruction scheduling is to produce a schedule which has minimum completion time when run on the processor. Note that the completion time for an instruction stream run on a processor with out-of-order execution may be shorter that the length of the schedule. This is caused by the fact that, in the presence of instructions executed out of order, idle slots may be "filled" with instructions from other basic blocks. Consequently, *globally optimal* schedules are not necessarily composed of *locally optimal* ones.

Our algorithm has as input a DAG (as explained above). The output from the algorithm is a local schedule for each basic block which is globally optimal for the trace. However, for clarity of presentation, we will only give the algorithm to compute the *cost* of such a schedule. It is straightforward to add code to keep track of the schedule itself.

The algorithm is based on "guessing" (enumerating rather) the nodes at the beginning and at the end of a basic block. Some "guesses" will permit speculative execution at a particular basic block boundary, while others will not.

The data structures used for the algorithm are:

- a list of records with information about the first two nodes of a schedule:
  first – the first node in the schedule
  succ – the second node in the schedule
  latency – the latency between them

  Head2, of this type, is used for the head of the current basic block

- a list of records with information about the last two nodes of a schedule:
  last – the last node in the schedule
  pred – the predecessor of *last* in the schedule
  latency – the latency between them
  cost – the cost of the schedule which ends in these two nodes

  Tail2, of this type, is used for the endings of the current basic block, while Seq_Tail2 is used for the sequence of basic blocks up to the current basic block.

The pseudocode for the main procedure is given in figure 1. We start with Seq_Tail2 consisting of a single record (corresponding to a single ending of the "previous" block). This is only a device for initializing the loop. The solution is iteratively extended for each basic block in the trace. At the end, it may be

```
procedure schedule_trace
(1)     Seq_Tail2 := {pred=0,last=0,latency=0,cost=0}
(2)     for each BB in the trace do
(3)         extend_sequence(Seq_Tail2, BB)
(4)     cost := Seq_Tail2[1].cost
(5)     for i in 2 .. #Seq_Tail2 do
(6)         cost := min( cost, Seq_Tail2[i].cost)
(7)     return cost
end
```

Figure 1: Optimal Trace Scheduling

```
procedure extend_sequence(Seq_Tail2, BB)
(8)     (Head2, #Head2) := make_heads(BB)
(9)     (Tail2, #Tail2) := make_tails(BB)
(10)    for h in 1 .. #Head2 do
(11)        min_head_cost := ∞
(12)        for t in 1 .. #Seq_Tail2 do
(13)            min_head_cost := min( min_head_cost,
                Seq_Tail2[t].cost + plug(Seq_Tail2[t],Head2[h]) - Seq_Tail2[t].latency - 2)
(14)        for t in 1 .. #Tail2 do
(15)            if consistent(BB, Head2[h], Tail2[t]) then
(16)                cost := min_makespan(BB, Head2[h], Tail2[t]) +
                            min_head_cost - Head2[h].latency - 2
(17)                Tail2[t].cost := min( Tail2[t].cost, cost)
(18)    Seq_Tail2 := Tail2
end
```

Figure 2: procedure *extend_sequence*

that the list Seq_Tail2 has more than one record, so we have to choose the one corresponding to minimum cost.

Procedure *extend_sequence* is presented in figure 2. It computes the cost of an optimum schedule of the trace including the current basic block, given the cost of all possible endings of the sequence up to the current basic block.

In line 10, the first two instruction are chosen. In lines 11–13 is computed the minimum cost of combining this particular beginning of the basic block with all possible endings of the previous sequence. In lines 14–17, we compute the cost of all possible endings of the current basic block compatible with the choice we have made for the beginning of it. The procedure *min_makespan* takes care of the modifications which need to be made to the graph BB in order for the schedule produced to have the head and tail as fixed (this can be achieved by

```
procedure make_heads(BB)
(19)    First := set of all nodes with no predecessors in BB
(20)    index := 1
(21)    for all h1 ∈ First do
(22)        for each successor h2 of h1 in BB do
(23)            Head2[index] := {first=h1,succ=h2,latency=latency(G,h1,h2)}
(24)            index := index + 1
(25)        for each h2 ∈ First - {h1} do
(26)            Head2[index] := {first=h1,succ=h2,latency=0}
(27)            index := index + 1
(28)    return {Head2, index-1}
end

procedure make_tails(BB)
(29)    Last := set of all nodes with no successors in BB
(30)    index := 1
(31)    for all t1 ∈ Last do
(32)        for each predecessor t2 of t1 in BB do
(33)            Tail2[index] := {last = t1, pred=t2,latency=latency(G,t2,t1),cost=∞}
(34)            index := index+1
(35)        for each t2 ∈ Last - {t1} do
(36)            Tail2[index] := {last=t1,pred=t2,latency=0,cost=∞}
(37)            index := index+1
(38)    return {Head2, index-1}
end
```

Figure 3: Generating Heads and Tails

introducing extra edges). The information accumulated in Tail2 is that required for a call to this procedure with the next basic block in the trace.

The code for computing all the possible beginnings of a basic block is given in figure 3. In line 19, we compute the set $First$ of all nodes with no predecessors. Choose the first node $h1$ to be one of them. Then, the second node $h2$ can be any of the other nodes in $First$ (in which case the latency between them will be 0), or a proper successor of $h1$ (in which case it retains the existing latency).

Similarly, for tails we compute the set *Last*, of all nodes with no successors in BB; $t1$ can be any of these nodes. Then, t_2 is either a proper predecessor of $t1$ or any of the remaining nodes in *Last* (the latency between $t2$ and $t1$ will be 0). The procedure *min_makespan* (figure 4) computes the minimum makespan of the basic block BB given the first and last two nodes. The underlying algorithm used for scheduling a basic block is that presented in [PS93], henceforth called *PS*. However, our algorithm remains correct with any other algorithm computing an optimum (local) schedule. Since the *PS* algorithm does not deal directly with fixed positions of some nodes in the schedule, we resort to modifying the graph,

```
procedure min_makespan(BB, H, T)
(39)    remove H.h1 from BB
(40)    remove T.t1 from BB
(41)    add edges with latency 0 from H.succ to all other nodes with no predecessor
(42)    add edges with latency 0 to T.pred from all other nodes with no successor
(43)    return makespan(BB) + H.latency + T.latency + 2
end
```

Figure 4: Procedure *min_makespan*

```
procedure plug(T, H)
(44)    G := restrict program graph to nodes {T.pred, T.last, H.first, H.succ}
(45)    add an edge with latency 0 from T.pred to H.first
(46)    add an edge with latency 0 from T.last to H.succ
(47)    return makespan(G)
end

procedure consistent(BB, H, T)
(48)    return ({H.h1, H.h2} ∩ {T.last, T.pred} = ∅) or
        ( number_of_nodes(BB) = 3 and H.first ≠ T.last)
end
```

Figure 5: Auxiliary procedures

by adding edges, in order to obtain such a schedule.

The auxiliary procedures *plug* and *consistent* are presented in figure 5. Procedure *plug* computes the execution time for a schedule with the first two instructions being those of T and the last two instructions being those from H. This is the place where speculative execution may occur.

Procedure *consistent* returns true in case that the choice of the last two nodes is consistent with that of the first two nodes in a basic block.

4.1 Analysis

It can be seen that the resulting algorithm is (time-wise) linear in the length of the trace. If the largest basic blocks has size n, the algorithm has a worst case complexity of $O(L \times n^4 \times PS)$, where PS is the time taken by the Palem-Simons algorithm to schedule one basic block[3]. The n^4 factor comes from a very loose bound on the number of pairs heads-tails a graph can have: there can be at most $\binom{n}{2}$ heads (or tails), giving rise to n^4 head-tail pairing possibilities. Also, each

[3]Excluding the time to compute the transitive closure of the graph.

head has to be tried with the tails of the preceding sequence of basic blocks, of which there can be at most $\binom{n}{2}$.

Although a number of improvings can be made to the above algorithm to have a better expected time, we will not attempt to describe them here.

5 Conclusion

We were able to find a polynomial algorithm for $s = 2$. It can be seen that the algorithm is easily generalizable to greater values of s. However, that brings a corresponding increase in the time complexity of the algorithm. Based on our experience with this problem, we conjecture that an algorithm to compute the optimum schedule for arbitrary values of s (s part of the input) is NP-hard. We are currently working on proving this.

6 Acknowledgments

I would like to thank Krishna Palem and Allen Leung for the helpful discussions we had about scheduling algorithms for processors with lookahead. It is our work together in assessing the benefit of fast heuristic algorithms that got me interested in searching for an algorithm to produce an optimal schedule.

References

[LPU95] Allen Leung, Krishna Palem, and Cristian Ungureanu. Run-time versus compile-time instruction scheduling in superscalar (risc) processors: Performance and tradeoffs. Technical Report TR-699, New York University, July 1995.

[PS93] Krishna Palem and Barbara Simons. Scheduling time-critical instructions on RISC machines. *ACM TOPLAS*, 5(3), 1993.