List Scheduling: Alone, with Foresight, and with Lookahead

Steven J. Beaty
Cray Computer Corporation
1110 Bayfield Drive
Colorado Springs, Colorado, 80906
Voice: (719) 540-4129
FAX: (719) 540-4028
beaty@craycos.com

Abstract

List scheduling is a popular method of scheduling. It has the benefits of being a relatively fast technique and producing good results. It has weaknesses when dealing with restricted timing. Two ancillary methods, foresight and lookahead, have been developed to help mitigate this weakness. This paper compares the effectiveness of list scheduling alone, with foresight, and with lookahead. It also shows the benefits of lookahead are accrued with no time overhead.

1 Introduction

List scheduling (LS) is a general scheduling method [Cof76] often used for both instruction scheduling (IS) [Gas89] and processor scheduling (PS) [WSG92]. While the techniques discussed in this paper are demonstrated using IS, they extend to other forms of scheduling. LS builds a *ready set* that contains all jobs that are not waiting on the results of another job. The ready set is then heuristically ordered, and the highest priority node is placed in the final schedule. This process repeats until there are no more nodes to place. In finding the ready set, LS performs a topological sort of the directed acyclic graph (DAG), thereby reducing the search space of the scheduling problem and increasing the chances of finding a valid schedule. List scheduling has an implicit heuristic: scheduling nodes with no predecessors produces valid orderings more often than scheduling nodes with predecessors. As with all heuristics, there are instances where this assumption does not hold.

IS involves the placement of atomic machine operations into machine instructions. A data dependence DAG (DDD) is often used to describe the necessary operations and their order. The nodes in a DDD contain the operations, and the edges denote a partial order on the nodes. This partial order is used to guarantee program dataflow requirements. The edges of a DDD do not constrain the order nodes are scheduled, only the order they appear in the final schedule.

In [Veg82], Vegdahl uses both minimum and maximum times on the edges in a DDD to express complicated timings between nodes. In the current paper, $\Delta(e) = (min, max)$ will be used to denote the timing associated with an edge in a DDD. This allows the description of a rich set of architectural features, and importantly, allows the description of many different kinds of architectures in one representation. Therefore, a scheduler using this representation can be made generic and will work for any number of different machines. For IS the following are easily expressed with edges that have non-infinite maximum timing:

- multi-stage pipes, either homogeneous or heterogeneous (e.g., a single pipe that does both multiplications and additions),
- transient resources such as the latent register designation on I860 [Int90] pipe operations, and
- other operations extending beyond one clock cycle including delayed branches.

Resources that latch their values (such as general-purpose registers) are modeled with the maximum time set to an infinite value. For PS, semaphores and other interprocess[or] communication can be modeled using non-infinite maximum timings.

Using $\Delta(e)$, the range of instructions where each operation can be placed can also be calculated. This range will be termed $\Theta(op) = (min, max)$, meaning op can be scheduled in any instruction $\iota\{\iota \mid min \leq \iota \leq max\}$. This is termed the *absolute timing* [AM88] for op.

The absolute timing calculation provides an easy method to check for timing errors. For example, if a node's earliest time becomes later than its latest time, a timing error is present in the current DDD. This checking provides a means for detecting errors either in the order of packing nodes from the DDD or in the DDD itself. This checking has the same result as *extended timings* [SDX87] have.

2 LS Enhancements

Because LS is based on an implicit heuristic and uses other heuristics to order the jobs, it does not always generate valid schedules. Often, the node-ordering heuristics must be tuned for each particular architecture in order to improve LS's effectiveness. There are numerous difficulties with this, including:

- the human must understand the impact of the heuristics on the scheduling and understand how to tune them to increase the likelihood of valid schedules,
- this effort takes time when retargeting to a new architecture,
- the heuristics that help guarantee validity may lead to longer schedules unnecessarily and,
- all the heuristic tuning still does not guarantee validity.

An example follows to illustrate when LS can fail during scheduling. Figure 1 is an actual DDD taken from ROCKET [SB92], a highly-optimizing retargetable compiler, targeted for the IBM RS/6000 [IBM90]. The code in the block's nodes is displayed in Figure 2. This code is the inner part of the following loop:

```
for (i = 0; i < 15; i++)
        up[i] = down[i] = 1;
```

The assembly code for nodes 6, 7, 12, and 13 is empty. These nodes are used to synchronize the store operations, each of which takes two machine cycles. When scheduling a DDD, one can either go top-down or bottom-up. In architectures with delayed branches, ROCKET usually goes bottom-up as the placement of the branch and surrounding nodes is less restricted by previous placement decisions. Note however that any situation found going in one direction can be replicated in the other, so direction in unimportant when considering the generation of valid schedules. Using plain LS, ROCKET chooses nodes 13, 12, 7, 11, 6, 9, 8, 10, ... as the order of placement. A difficulty arises when node 7 is placed in instruction 3 as this constrains node 6 which constrains 11 and 5, which constrains 10 and 8, which constrains 9. As node 5 now must be placed in instruction 5, nodes 8 and 10 must be either in instruction 4 or 5. Operations 5, 8, and 10 all need the integer unit of the processor so all three cannot fit into the two instructions.

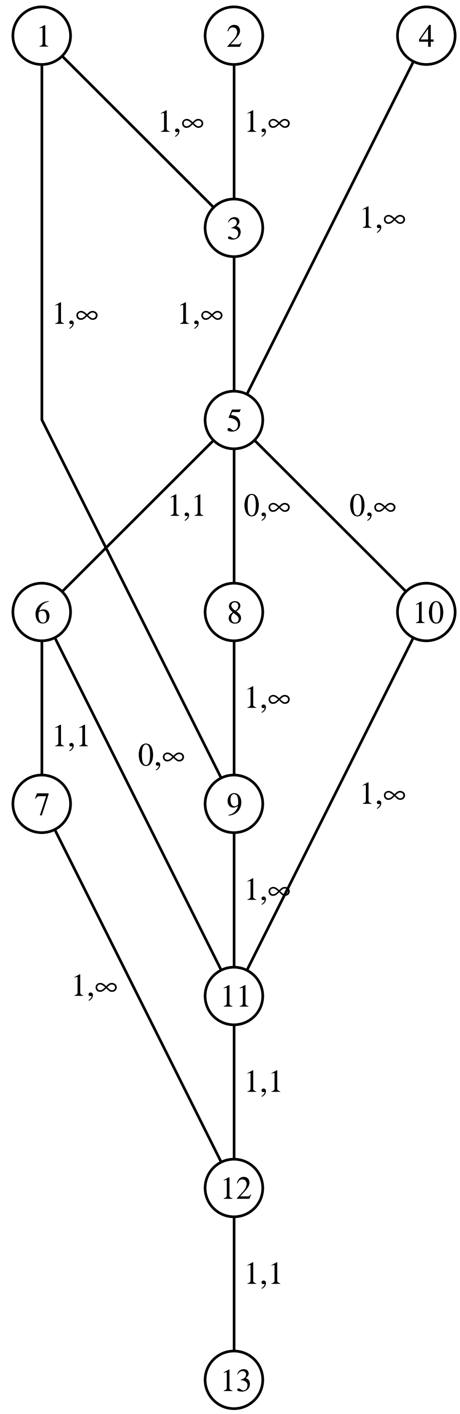

Figure 1: A DDD where List Scheduling can fail

Node	Code
1	sli r15,r13,2
2	cal r14,G_down(30)
3	a r14,r15,r14
4	lil 31,1
5	st 31,0x0(r14)
6	# empty
7	# empty
8	cal r14,G_up(30)
9	a r14,r15,r14
10	lil 31,1
11	st 31,0x0(r14)
12	# empty
13	# empty

Figure 2: Code for nodes

2.1 Foresight

A powerful method to increase the likelihood of generating a valid schedule called *CAS* (for Check And Schedule) was introduced by Su [SDX87]. This method checks to see if all successors of a node being considered for placement in an instruction can be placed in their respective instructions. If not, the operation is not placed in the current target instruction. Allan et al. [ASWW92] extended CAS to include all nodes in the graph with non-infinite maximum absolute times and renamed it *foresight*. Foresight checks to see whether, after placing an operation in an instruction, all nodes that become constrained (having $\Theta(op) = (a,b)\{b \mid b < \infty\}$) can be "easily" placed in their respective instructions with respect to resource conflicts. If so, the operation under consideration is placed. If not, the operation is moved to its next valid instruction and foresight is repeated. If no valid instruction can be found, the schedule generated thus far is deemed invalid. Because a substantial amount of information is generated during each pass of the foresight routine, Wijaya and Allan [WA89] added the ability to keep information from one pass to another, resulting in *incremental* foresight. This is possible because the schedule ranges for operations have temporal locality; i.e., once they are constrained, they remain constrained.

Note that foresight can correctly schedule the graph in Figure 1. It decides not to place node 5 in instruction five, instead finding that it must be placed in instruction seven in order to create a valid schedule. While foresight greatly increases the chances of generating valid schedules, it also adds to the time required to perform the scheduling. In [WA89], four schedules that failed during LS were scheduled using foresight. An average time increase of 65% was noted for non-incremental foresight, and 26% for incremental foresight.

2.2 Lookahead

As noted before, the edges in a DDD only limit the ordering in the final schedule, not the order the schedule is created. So long as the partial order is preserved, the order of placing the nodes is irrelevant. The value $\Theta(op)$ for a node, calculated by the absolute timing routine, specifies the range in the final schedule where an operation can be placed. Because the foresight routine examines instructions in this range for node placement, if foresight succeeds in finding a valid place for an operation, then that placement will be valid in the final schedule. An alternative view is that not only **can** a node be placed where foresight predicts, it **should** be placed there. A method termed *lookahead* [Bea92] was developed to place operations instead of just testing for the possibility of placement. The original motivation for lookahead was to increase both the speed and the chances of creating valid schedules for a stochastic scheduling method [Bea91]; it was then noticed it could speed up generic LS as well. The remainder of this paper will explore the implications of using lookahead in LS and also discuss the performance of the algorithm.

Several minor changes to LS with lookahead need to be noted. First, the definition of data ready does not change; i.e., it is still those nodes in the graph that have no unscheduled predecessors. The computation of these nodes might be different. It is no longer enough to remove nodes from the data ready set when they are placed by LS; one must also add and remove nodes based on those lookahead places. Lookahead can remove any or all the nodes on the data ready set; it can also make nodes further down in the graph data ready by placing all their predecessors. The scheduler must also ignore all the nodes that are placed by lookahead during later stages of the scheduling process. Both of these conditions are handled in the compiler by the addition of a flag in the nodes that state whether or not the node has been placed, either by LS or by lookahead. It is also important for lookahead to check nodes in a breadth-first manner so that no cycles develop during the procedure.

Note that any failure to placing a restricted node with lookahead would result in a failure later in the scheduling process, thereby reducing the time spent scheduling an infeasible schedule. Lookahead also schedules nodes without having to topologically order them which is a time consuming process. By doing so it reduces the number of nodes LS must deal with and thereby increases the speed of scheduling.

Also note that naive lookahead places nodes in the final schedule non-heuristically. That is, there is no order in examining the constrained nodes based on node weights built into lookahead. While this expedites the process, lookahead could be extended to deal directly with differing priorities in the constrained node set so that the final schedule length is optimized.

It is important to understand that using lookahead with list scheduling is still an avoidance technique. There still is the possibility that valid DDDs exist that cannot be scheduled due to poor choices made by the node priority heuristics. This is an inherent problem when only searching a small subspace of the possible solutions.

Name	Succeeded	Failed
8q	16	4
dhrystone	59	12
diff3	69	20
livermore	236	110
sort	34	9
whetstone	46	7

Figure 3: Failures of plain List Scheduling

3 Results

In order to compare LS, LS with foresight, and LS with lookahead, some studies were made. For all, the target architecture was the IBM RS/6000. This architecture was chosen as representative of today's level of superscalar design. The programs, all written in C, were a mixture of numerical analysis and general-purpose code: the 8q solves the 8 queens problem, dhrystone [Wei88] is an integer arithmetic benchmark, diff3 is a GNU 3-way file difference program, livermore is the 24-loop Livermore Loops [McM86], sort is a quicksort program, and whetstone [CB76] is a floating-point benchmark program.

In Figure 3, a table of scheduling failures for LS without foresight or lookahead is shown. The numbers represent the number of basic blocks that were either scheduled, or that LS failed to properly schedule. The heuristics that drove LS emphasized the number of successors, the number of restricted successors, the height in the DDD, and whether an operation is on the critical path for the block. This combination of characteristics has been effective in the past for generating good, valid schedules. The rate of failure is rather high, pointing to the fact that these heuristics are not enough, and that most other schedulers must use ad hoc methods to guarantee validity. Once foresight or lookahead was added, no failures to schedule were found. This points to the power of these two methods. As an additional test, all heuristics were removed from the scheduler; again no failures to schedule were found. This implies that performing foresight or lookahead is more important for forming valid schedules than choosing good heuristics. Good heuristics are important only when valid schedules can be guaranteed and schedule length becomes the overriding issue.

Figure 4 shows the running time in milliseconds of the scheduling routines for each program. The times are for a lightly-loaded Sun SS10/30. Timings for foresight are not included in the figure as there was no need to implement the more complicated incremental foresight algorithm in ROCKET. ROCKET implements foresight by not placing the constrained nodes during lookahead. There is no temporal locality associated with lookahead as it places all constrained nodes in one pass and can therefore ignore those nodes completely in subsequent passes. Figure 5 contains the number of operations lookahead scheduled and the total number of operations scheduled.

Name	LS	Lookahead	%
8q	912	1034	113.38
dhrystone	5326	4327	81.24
diff3	4642	4604	99.18
livermore	26613	32436	121.88
sort	2680	2150	80.22
whetstone	8353	9830	117.68
Average			102.26

Figure 4: Timing Comparison

The speed of lookahead is heartening. It is, on the average, 2% slower that plain LS and $\approx$ 20% faster than incremental foresight. It does require more analysis on the DDD, but makes up for it by placing operations immediately when they become constrained, removing the number of nodes LS must examine. In these programs, lookahead placed an average of 54% of the nodes in the graph, with a range from 48% (in 8q and sort) to 62% (in livermore.) Lookahead is noticeably faster than incremental foresight. This disparity is larger when more operations having restricted timing are present in the target architecture, and when programs use those operations.

Name	Placed	Total	%
8q	80	165	48.48
dhrystone	401	739	54.26
diff3	375	788	47.59
livermore	3094	5000	61.88
sort	212	388	54.64
whetstone	817	1472	55.50
Average			53.73

Figure 5: Number of nodes lookahead placed

4 Conclusions

List scheduling is a good method for instruction scheduling for most architectures, especially when combined with methods that check for the placement of constrained operations. Foresight and lookahead are two methods that do this. Lookahead is able to greatly enhance list scheduling's ability to generate valid schedules, at essentially no cost. As architectures are developed that contain more available parallelism, with more constraints on the timing between operations, lookahead will become more and more important.

References

[AM88] V.H. Allan and R.A. Mueller. "Microcode compaction with general synchronous timing". *IEEE Transactions on Software Engineering (Special Section on Microprogramming)*, 14(5):595–599, May 1988.

[ASWW92] Vicki H. Allan, Bogong Su, Pantung Wijaya, and Jian Wang. "Foresighted compaction under timing constraints". *IEEE Transactions on Computers*, 41(9):1169–1172, September 1992.

[Bea91] S.J. Beaty. *Instruction Scheduling Using Genetic Algorithms*. PhD thesis, Mechanical Engineering Department, Colorado State University, Fort Collins, Colorado, 1991.

[Bea92] Steven J. Beaty. "Lookahead scheduling". In *Proceedings of the 25th Annual International Symposium on Microarchitecture (Micro-25)*, pages 256–259, Portland, Oregon, December 1992.

[CB76] H.J. Curnow and Wichman B.A. "A synthetic benchmark". *Computer*, February 1976.

[Cof76] E.G Coffman. *Computer and Job-Shop Scheduling Theory*. Jon Wiley & Sons, New York, 1976.

[Gas89] F. Gasperoni. "Compilation techniques for vliw architectures". Technical report, Courant Institute of Mathematical Sciences, New York University, March 1989.

[IBM90] IBM. *IBM Journal of Research and Development*, January 1990.

[Int90] Intel. *i860 64-bit Microprocessor Programmer's Reference Manual*, 1990.

[McM86] F.H. McMahon. "The livermore fortran kernels: A computer test of numerical performance range". Technical report, Lawrence Livermore National Laboratory, December 1986.

[SB92] Philip H. Sweany and Steven J. Beaty. "Rocket retargetable c compiler – an overview". Technical report, Archelon Inc., 460 Forestlawn Road, Waterloo, Ontario, Canada, N2K 2J6, 1992.

[SDX87] B. Su, S. Ding, and J. Xia. "Microcode compaction with timing constraints". In *Proceedings of the 20th Microprogramming Workshop (MICRO-20)*, Colorado Springs, CO, December 1987.

[Veg82] S.R. Vegdahl. *Local Code Generation and Compaction in Optimizing Microcode Compilers*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1982.

[WA89] P. Wijaya and V.H. Allan. "Incremental foresighted local compaction". In *Proceedings of the 22nd Microprogramming Workshop (MICRO-22)*, Dublin, Ireland, August 1989.

[Wei88] R.P. Weicker. "Dhrystone benchmark: Rationale for version 2 and measurement rules". *SIGPLAN Notices*, 23(8):49–62, August 1988.

[WSG92] Tia M. Watts, Mary Lou Soffa, and Rajiv Gupta. "Techniques for integrating parallelizing transformations and compiler based scheduling methods". In *Supercomputing '92 Proceedings*, 1992.