Lookahead Scheduling

Steven J. Beaty
Department of Mechanical Engineering
Colorado State University
Fort Collins, Colorado 80523
beaty@longs.lance.colostate.edu

Abstract

A new method of scheduling operations is presented. Lookahead scheduling, alone or in combination with other scheduling methods, can increase both the speed and the likelihood of correctness when generating schedules.

1 Introduction

Sequencing is defined by Ashour [Ash72] as being "concerned with the arrangements and permutations in which a set of jobs under consideration are performed on all machines." That is, what is the order the jobs will be performed; what is the priority of each job? Sequencing thereby ranks the jobs to be executed. Baker [Bak74] states "scheduling is the allocation of resources over time to perform a collection of tasks." Scheduling usually places already prioritized jobs into slots, often accounting for conflicts in resource usage. The combined sequencing/scheduling (order/place) process produces the desired outcome: jobs placed on machines capable of performing the desired tasks in the correct order at a correct time. In [Bea92], a more thorough treatment of the possible combinations of sequencing and scheduling is given.

Instruction scheduling (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.

List scheduling (LS) is a general [Cof76] scheduling method often used for instruction scheduling [Gas89]. LS builds a ready set that contains all jobs that are not waiting on the results of another job. In IS, this is represented as nodes with no unscheduled predecessors. In finding the ready set, LS performs a topological sort of a DDD, 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 results in valid orderings more often than scheduling nodes with predecessors. As with all heuristics, there are instances where this assumption does not hold.

Trace scheduling [Fis81] relies on a LS algorithm to schedule traces. In this way, it is really a meta-scheduling algorithm. It provides the raw material for another scheduler; it does not prioritize or place the operations. Percolation scheduling [Nic85] is both a scheduler and a meta scheduler. The set of allowed graph operations are used to schedule the machine operations; there are heuristics to prioritize which of the allowed graph operations are to be used.

2 LS Enhancements

Because LS is based on an implicit heuristic and uses other heuristics to prioritize the jobs, it does not always generate valid schedules. A number of different methods have been used to assist LS with the generation of good, valid schedules. This section discusses several.

2.1 Absolute Timing

$\Delta(e)$ is used to denote the timing associated with an edge in a DDD. It contains both a minimum and maximum time allowable between operations to describe a rich set of architectural features. With this definition on edges, timing can be assigned to the nodes as well. Using $\Delta(e)$, a range of instructions where each operation can be placed can 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* for op.

$\Theta(op)$ can have a large influence when calculating node priorities. Using the absolute timing algorithm from Allan and Mueller in [AM88], and Figure 1, node 3 will originally have $\Theta(3) = (2, \infty)$. After the placement of node 2, node 3 will have $\Theta(3) = (n, n)$ where n is one greater than the scheduled value of node 2. This is a much tighter

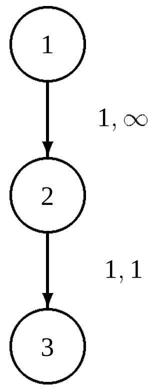

Figure 1: Need for dynamic schedule range calculation

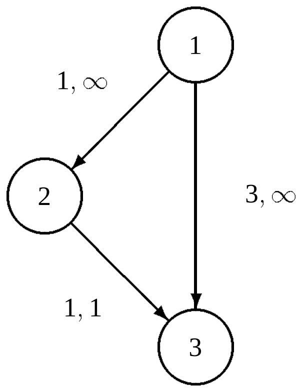

Figure 2: A graph where the absolute timing algorithm can fail

bound on the range of node 3 and should be reflected in its priority.

A complication not addressed in [AM88] is the possibility of legal loops in the timing. In Figure 2, if the absolute timing algorithm is run, the following occurs:

1. Starting with $\Theta(1) = (1, \infty)$: $\Theta(2) = (2, \infty)$, $\Theta(3) = (4, \infty)$.
2. Traverse to 3, change predecessors: $\Theta(2) = (3, \infty)$.
3. Traverse to 2, change predecessors: $\Theta(1) = (2, \infty)$. Goto step 1, creating a loop.

The difficulty occurs when the routine follows a $\Delta(e) = (n, \infty)$ edge to a previous node (from node 1 to node 2), changing its timing. An important observation is

> there is no reason to follow (n, ∞) edges to predecessors. The absolute timing algorithm is only interested in making the timing on nodes *later*, and with ∞ maximum timing on an edge, no reason exists to make a predecessor node later. The operation may occur as early as possible and the value will remain valid forever.

During the calculation of absolute timing, it is easy to check for timing errors. For example, if a node's earliest time becomes later than its latest time, a timing error with respect to its neighbors is present. It is also an error to change the timing to be earlier than it currently is. The node is as early as it can be in relation to some node; if another requires it earlier a timing error is present. This checking provides a means for detecting errors in the order of packing nodes from the DDD. This has the same result as *extended timings* [SDX87] have, although not necessarily as early in the scheduling process.

2.2 Foresight

A powerful method to increase the likelihood of generating a valid schedule called *foresight* is introduced by Su [SDX87]. The procedure checks to see whether, after placing an operation in an instruction, all nodes that become constrained (having $\Theta(op) = (a, a)$) can be "easily" placed in their respective instructions, with respect to resource and encoding 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. This is not a backtracking algorithm; on the contrary, it looks forward, checking for validity of placing a node before the final decision for any node is made. 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. The schedule ranges for operations have a form of temporal locality, i.e. once they are constrained, they remain constrained. The constrained set does not vary greatly between iterations. Rules may be formed that specify when schedule ranges are affected by placement of operations and must be updated. When incremental foresight cannot reuse information from a previous pass, non-incremental foresight is used. In this way speedup is achieved and incremental foresight fails only when foresight would.

Foresight certainly helps in the generation of valid schedules by checking the validity of operation placement before committing to it. An assumption made by foresight is either 1) constrained nodes will be placed in the instructions foresight checks, or 2) other instructions can be found to hold them. This assumption can be invalidated. If the constrained nodes cannot be placed in the instruction examined, foresight breaks down. For example, if instead of the constrained nodes being scheduled into the instructions foresight examined, nodes from another part of the DDD are scheduled into those instructions (either due to higher priority or the successor nodes not being data ready), the validity of the examination is obviated.

3 Lookahead

When the scheduler does not place the constrained nodes in the instructions that foresight determined will produce a valid schedule, work is lost. Why not perform the scheduling of those nodes immediately? Within the framework of list scheduling, the reason is simple: those nodes may not be data ready. If the data ready criterion is removed, what is the impact upon forming a valid schedule? None.

As noted before, the edges in a DDD only limit the ordering in the final schedule. So long as this order is preserved, the method of placing the nodes is irrelevant. The checking done by the absolute timing algorithm assures that nodes are placed such that their range is valid in the final schedule. 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 schedule 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. If it is not, scheduling can fail on a placement it previously judged valid by foresight. A method termed *lookahead* was developed to place operations instead of just testing for the possibility of placement.

As a simple example of lookahead, consider the DDD in Figure 1. When attempting to schedule node 2, node 3 becomes completely constrained. Foresight checks to see if node 3 can be packed in the given instruction. If it can, node 2 is placed and scheduling continues. Lookahead also checks node 3, if it can be placed both node 2 and node 3 are scheduled, guaranteeing scheduling will not fail later due to the inability to place node 3. When foresight or lookahead fail, node 2 will not be placed in the original instruction.

A decision must be made as to whether to pack only the nodes with $\Theta(op) = (a, a)$ (equivalently $\Delta(e) = (n, n)$), or additionally to pack the nodes with $\Theta(op) = (a, b)\{a, b \mid a < b < \infty\}$. In the first case, no choice exists as to when to pack the nodes, they must be placed in instruction $\iota + n$. The second case contains more flexibility and requires the analysis of a tradeoff. Having lookahead place them will result in a larger chance of generating a valid schedule, similar to the improvement that foresight has to plain list scheduling. However, if lookahead does not immediately place the $\Theta(op) = (a, b)\{a, b \mid a < b < \infty\}$ nodes, the scheduler may be able to produce a more compact final sequence.

This tradeoff varies with the amount of flexibility in the operation's schedule range, in the current DDD, and in the architecture, making it difficult to analyze the tradeoff universally. For example, if the current DDD is wide (displaying a lot of parallelism), constrained operations might need to be placed immediately so that other parallel operations do not consume all needed resources in $\Theta(op)$. For machines with a large amount of available parallelism, final placement should probably be deferred, allowing the most amount of flexibility for the scheduler. Placement decisions made between the time of finite constraint and final packing are less likely to have a deleterious effect as there is more "room" in each instruction for operations in these types of architectures. A heuristic based on $\Gamma(op) = \Theta(op)_{max} - \Theta(op)_{min}$ could be tuned on a per-machine basis to control the amount of lookahead.

3.1 With List Scheduling

Lookahead can be combined with LS to increase both the chances of generating a valid schedule and the speed by which the schedule is generated. There is no need to change the definitions or implementations of any routines within the list scheduler. The definition for data ready remains the same. Any failure to place a restricted node would result in a failure later in the scheduling process, reducing the time spent on an infeasible schedule. Lookahead also schedules nodes without having to topologically order them. By doing so it removes the number of nodes LS must deal with and thereby increases the speed.

It is important to understand that using lookahead with list scheduling is still an avoidance technique, albeit a more powerful one than foresight, itself more powerful than nothing at all. 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.

3.2 Without List Scheduling

Because lookahead is a scheduling method unto itself, it may be combined with other methods of sequencing. The sequencing method simply provides the order of placement of the operations and lookahead schedules them. An example of combining lookahead and genetic algorithms may be found in [Bea91]. A difficulty with relying on lookahead to schedule is it is a relatively "weak" scheduling method as it uses no heuristics to attempt to produce valid schedules. The method of ordering the placement must therefore be stronger. Certainly genetic algorithms is one such method. A benefit of using lookahead is its speed. For example, if a sink node in a DDD is scheduled first, all others may be packed in one step by lookahead. This occurs because the absolute timing algorithm constrains all other nodes to a specific instruction. This assumes of course that this constraining produces a valid schedule. If not, iteration on the sequence may be necessary.

4 Conclusions

A new method of scheduling operations, lookahead, has been presented. Lookahead may be used in conjunction with other scheduling techniques such as list scheduling to increase both the likelihood of generating correct schedules and the speed the production of them. Lookahead may also be used with other methods of sequencing to form a complete scheduling system.

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.

[Ash72] S. Ashour. *Sequencing Theory*. Springer-Verlag, New York, 1972.

[Bak74] K. R. Baker. *Introduction to Sequencing and Scheduling*. John Wiley and Sons, Inc., New York, 1974.

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

[Bea92] S.J. Beaty. "Genetic algorithms for instruction sequencing and scheduling". In *Proceedings from the Workshop on Computer Architecture Technology and Formalism for Computer Science Research and Applications*, Istituto per le Ricerche sui Sistemi Informatici Paralleli via P. Castellino 111, 80125 Napoli (Italy), March 1992.

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

[Fis81] J.A. Fisher. "Trace scheduling: A technique for global microcode compaction". *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

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

[Nic85] Alexandru Nicolau. "Percolation scheduling: A parallel compilation technique". Technical report, Department of Computer Science, Cornell University, Ithaca, New York, May 1985.

[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.

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