Efficient DAG Construction and Heuristic Calculation for Instruction Scheduling

Mark Smotherman[1] *Sanjay Krishnamurthy*[2] *P.S. Aravind*[1] *David Hunnicutt*[1]

[1]Department of Computer Science
Clemson University
Clemson, SC 29634-1906

[2]Department of Computer Science
Rice University
Houston, TX 77251-1892

ABSTRACT

A number of heuristic algorithms for DAG-based instruction scheduling have been proposed over the past few years. In this paper, we explore the efficiency of three DAG construction algorithms and survey 26 proposed heuristics and their methods of calculation. Six scheduling algorithms are analyzed in terms of DAG construction and heuristic use. DAG structural statistics and scheduling times for the three construction algorithms are given for several popular benchmarks. The table-building algorithms are shown to extremely efficient for programs with large basic blocks and yet appropriately handle the problem of retaining important transitive arcs. The node revisitation overhead of intermediate heuristic calculation steps is also investigated and shown to be negligible.

1. INTRODUCTION

Instruction scheduling is the transformation of intermediate code, assembly language statements, or machine code to increase performance. This transformation often involves the reordering of instructions to eliminate or reduce the number of stalls encountered on a pipelined computer. Reordering and instruction substitution (or migration) can also be used to provide a more balanced instruction stream to the multiple function units of a superscalar processor. However, all transformations are constrained by dependencies between the instructions, which can be classified as data, control, or structural dependencies.

Data dependencies include read-after-write (RAW or true dependency), write-after-read (WAR or anti- dependency), and write-after-write (WAW or output dependency). In order to maintain the semantic correctness of a program, transformations must preserve data dependencies. Control dependencies result from branch and call instructions. Delayed branching and branch prediction are often used to reduce the effect of this type of dependency. Structural dependencies exist when two instructions require the same function unit for execution; thus, one instruction must stall or the function unit must be replicated or pipelined.

One of the first experiments dealing with increasing performance by reordering instructions was done by Sites for the Cray-1 supercomputer [14], and the importance of reducing stall conditions on IBM mainframes was noted by Rymarczyk [11]. More recently, literature on instruction scheduling has focused on techniques at the basic block level for pipelined and superscalar microprocessors [3,6,8,12,13,15,16]. New techniques are being developed to extend beyond basic blocks (e.g., see [1]); however, good basic block scheduling techniques are important components of extended techniques.

Scheduling at the basic block level consists of three steps. First, the data dependencies are used to construct a DAG (directed acyclic graph), in which the instructions are represented by nodes and dependencies are represented by arcs. Structural hazards are not represented in the DAG because they are essentially undirected arcs; instead, they are handled by timing heuristics or resource reservation tables. Control hazards can also be handled in a special manner, possibly by a delay slot scheduler.

The nodes and/or arcs of the DAG are annotated with information necessary for the scheduling pass to choose between available instructions. While some information related to these heuristics can be determined during the DAG construction step, not all can be obtained when the DAG is constructed in an order that is reversed with respect to the calculation of some static heuristics or when some heuristics are dynamic and depend on the scheduling order. For example, if the DAG is constructed in a forward pass, then a heuristic such as maximum path length from each node to a leaf node must be calculated by an additional backward pass over the DAG. Thus, there is often a second step in instruction scheduling, which is an intermediate pass over the DAG in the

opposite direction of DAG construction (e.g., [8,13,16]).

Finally, the actual scheduling is performed as a pass made over the DAG. Instructions are chosen according to some form of list scheduling or critical path algorithm, since finding the optimal order is an NP-complete problem [6]. List scheduling algorithms examine a candidate list of ready-to-execute instructions at each time step and apply one or more heuristics to determine the "best" instruction to issue. Critical path algorithms try to identify those instructions that must be executed as early as possible; that is, if any of the instructions on the critical path are delayed, the execution time will be correspondingly lengthened. Of course, heuristics indicating the critical path are often used for list scheduling.

Avoiding structural hazards is often done by ad-hoc methods or by use of an "earliest start time" heuristic. A more refined form of scheduling uses an explicit resource reservation table and is more popular for use with processors having a large number of multi-cycle instructions or multiple resource usage instructions. This latter approach always inserts the "highest priority" instruction into the earliest empty slots of the table; that is, an instruction is a aggregate structure represented by blocks of busy cycles for one or more function units, and scheduling involves pattern matching these blocks into a partially-filled reservation table as well as considering operand dependencies.

Many heuristics have potential use for instruction scheduling, and they can be classified into six broad categories: stall behavior, instruction class, critical path, uncovering, structural, and register usage. Stall behavior and instruction class heuristics attempt to avoid stall cycles, and critical path heuristics identify instructions that must be scheduled as early as possible. Uncovering heuristics try to enlarge the candidate list, while structural heuristics help balance progress through the DAG. Finally, register usage heuristics are used for pre-register-allocation scheduling to decrease the number of simultaneously live variables.

The next section reviews DAG construction issues and is followed by a detailed survey of different heuristics.

2. DAG CONSTRUCTION

A DAG is a directed acyclic graph consisting of nodes and directed arcs. A given node is said to be a *parent* of another node if there exists between them an RAW or WAW dependency on a resource defined by the given node or a WAR dependency on a resource used by the first node; conversely, the other node is described as the *child* of the given node. An *ancestor* of a node Y, is a node X such that parent-to-child arcs can be followed from X to Y. A *descendant* of a node is the converse: Y is a descendant of X if parent-to-child arcs can be followed from X to Y. A *leaf* node is a node with no children. A *root* node is a node with no parents.

DAGs are constructed for one basic block at a time. Along with branches, procedure calls may also mark the end of a basic block unless there are dependency arcs between the resources used and defined by the call and the instructions in the calling block (e.g., see the discussion in [16]). Register saving conventions may provide some flexibility in moving instructions across calls. Also, a consideration in register window machines (e.g., SPARC) is that register window alteration instructions (SAVE and RESTORE) mark the end of a basic block, since register identifiers name different physical resources on different sides of these instructions.

A basic block may result in a collection of one or more DAGs, called a *forest*. Some construction algorithms connect all DAGS in a forest by using a unique dummy root node as the parent of all true roots; typically this is done for convenience in representing an initial candidate list for a forward-pass scheduling algorithm. Additionally, some algorithms use a unique dummy leaf node or connect all true leaves to the block-ending branch node to ensure that the branch is the last node to be scheduled (e.g., see [16]). If global information (i.e., across basic blocks) is considered, there may be pseudo-nodes and arcs to represent operation latencies inherited from immediately preceding blocks. This extra information can be used to avoid dependency stalls and structural hazards that a purely local algorithm would ignore.

The DAG can be constructed either in a forward pass or a backward pass. Independent of the direction, the determination of whether a new node should be connected by dependency arcs to other nodes in the partially constructed DAG can be done by compare-against-all or by table building. Compare-against-all is an O(n**2) approach in which the new node is compared against all previous nodes. Table building is an approach that keeps a record of the last definition of a resource and the set of current uses. Backward-pass table building is [7]:

```
/* process resources defined */
 if ( resource[definition_entry] not empty and
      resource[uselist] is empty )
   add_arc(WAW, newnode, resource[definition_entry]);
 foreach ( uselist_entry in resource[uselist]
      in ascending order ) do {
   add_arc(RAW, newnode, uselist_entry);
   delete uselist_entry from resource[uselist];
 }
 insert newnode as resource[definition_entry];
/* process resources used */
 if ( resource[definition_entry] not empty)
    add_arc(WAR, newnode, resource[definition_entry]);
 add newnode as a uselist_entry into resource[uselist];
```

The forward-pass version is similar, but with resource uses processed before definitions [7,8].

The resources on which the data dependencies are determined include general registers, special purpose registers (e.g., condition codes), and memory locations. Register definitions and uses are straightforward to recognize, but there is sometimes not enough information after compilation to disambiguate memory references. The DAG construction algorithm may have to treat memory as a single resource, which leads to serialization of all loads and stores. It has been observed that if two memory references use the same base register but different offsets, they cannot refer to the same location. While this observation provides more flexibility than strict serialization, use of multiple base registers is common and references using different base registers must still be serialized; however, Warren noted that storage classes (e.g., heap vs. stack) typically do not overlap and that base registers for these different areas can sometimes be identified [16]. Even more sophisticated memory disambiguation methods have been developed for handling array references in vectorizing and parallelizing compilers (e.g., see [18]).

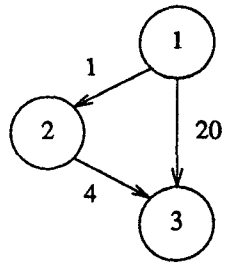
The arcs in the DAG are typically weighted according to operation latency; however, these latencies can differ according to the dependency type. For example, WAR delays can potentially be shorter than RAW delays because the the parent instruction reads (uses) the resource in an early pipeline stage. Care must be exercised when using this approach because on some machines source registers must be available well into the operation so that exception handlers can retrieve source values for use in fixing a faulted instruction.

Note that, from the same parent, different RAW delays can occur to different children. A simple example of this occurs on a double-word floating point load to a register pair; the RAW delays for these registers can be one or two cycles different. In the presence of operand bypass/forwarding, it may be the case that an RAW delay to an arithmetic operation may be longer than an RAW delay to a store operation. A more subtle example arises in the case of asymmetric bypass/forwarding paths (e.g., IBM RS/6000 [16]) in which an RAW delay for a given destination register to an instruction using that register as its first source operand will differ from the RAW delay to another instruction using that same register but as its second source operand.

A *transitive* arc is a parent-to-child connection between two nodes that also have an indirect ancestor-to-descendant connection through intermediate nodes. The n**2 approach has a huge number of these transitive arcs. In contrast, the table building methods omit most transitive arcs because they erase all but the most recent definition/uses. Indeed, some DAG construction algorithms are designed to prevent all transitive arcs. For example, the algorithm presented by Landskov, *et al.* [10], is a modification of the n**2 forward algorithm; it examines leaves first and prunes away any ancestors whenever a dependency is observed. Another approach to prevent transitive arcs is the use of reachability bit maps in backward-pass DAG construction. These maps use one bit position per node to indicate descendants. Each node's map is initialized to indicate that a node can reach itself. The algorithm for inserting an arc when a dependency is found is:

```
/* try to add arc from_a to to_b */
 if ( bit to_b in bitmap_for_a is set ) return;
 bitmap_for_a = bitmap_for_a OR bitmap_for_b;
 add_arc(from_a, to_b);
```

In considering whether to remove all transitive arcs, we note that transitive arcs can bias certain heuristics. For example, the number of children is artificially increased by each transitive arc. On the other hand, the use of small delays for WAR arcs is a strong argument against removal. Figure 1 contains an example DAG with a WAR-then-RAW path connecting three nodes and also a transitive RAW arc from the first node to the last. Because of the small latency of the WAR arc, calculation of some static as well as dynamic heuristics will be incorrect if the transitive arc is removed (e.g., sum of arc weights from node 1 to 3, earliest execution time for node 3). The table building methods discussed above will retain this kind of arc.



1: DIVF R1,R2,R3
(R3 = R1/R2, 20 cycles)

2: ADDF R4,R5,R1
(R1 = R4+R5, 4 cycles)

3: ADDF R1,R3,R6
(R6 = R1+R3, 4 cycles)

arc from 1 to 2 is WAR for R1 with delay of 1 cycle
arc from 2 to 3 is RAW for R1 with delay of 4 cycles
arc from 1 to 3 is RAW for R3 with delay of 20 cycles

Figure 1. Importance of transitive arcs

3. HEURISTICS FOR SCHEDULING

Table 1 shows various instruction scheduling heuristics organized according to broad classifications discussed in the introduction. Some heuristics are relationship-based, in which timing considerations are either not present or implicit. Others explicitly consider timing of the operations. This distinction is used to separate the heuristics into two columns. Also, a third column summarizes how the heuristics are calculated. The entries "a,b,f" in the

third column indicate static heuristics, which can be calculated prior to the scheduling pass; the entry "v" indicates a dynamic heuristic, which can only be calculated by node visitation during scheduling.

Details about the heuristics listed in Table 1 follow:

interlock with previous instruction - This predicate indicates whether a candidate node will be unable to execute in the next cycle due to a data dependency with the most recently scheduled node. It must be evaluated dynamically in a forward scheduling and requires repeated resource definition/use comparisons between the most recently scheduled instruction and the entire candidate list, or adding child-to-parent links to the DAG and following each of these links for the entire candidate list to see if a parent matches the most recently scheduled node and if the corresponding parent-to-child arc has a delay greater than one. Note that instructions scheduled earlier than the most recent but that have long latency dependencies or structural hazards are not considered, even though these may cause interlocks. This is an expensive heuristic, and its function is much better performed by earliest execution time.

earliest execution time - This is a dynamic heuristic that is calculated during a forward scheduling pass. A current time is maintained by the scheduler, and when an instruction is chosen each child has its earliest execution time updated by taking the maximum of the previous value and the current time plus the arc delay from the scheduled node. Nodes are admitted to the candidate list when all parents are scheduled and the earliest execution time is less than or equal to the current time. Note that this measure may be inaccurate when all transitive arcs are removed. If the function units are not pipelined, then structural hazards can be considered by performing a maximum earliest starting time calculation that includes the finish times of any required function units.

interlock with child - This static predicate indicates whether any child of a candidate node will be unable to execute in the next cycle, e.g., a load instruction with a one-cycle delay slot will have an RAW delay of two cycles with children that use the loaded value. The use of this heuristic in a forward scheduling pass attempts to choose instructions with long delays first, so that the remaining instructions in the candidate list can be used to fill the delay slots. It can be initialized as false and then set to true whenever the add_arc procedure is called with an arc delay greater than one.

execution time - This is the operation latency of the node. Use of this static heuristic attempts to choose instructions with long execution times first.

alternate type - This static heuristic attempts to alternate instruction selection among the different classes of instructions on a superscalar processor. That is, the instruction scheduler attempts to reorder the instruction stream so that as many instructions as possible can be issued each cycle. Indeed, an instruction scheduler may attempt to migrate instructions of one class to another class so that several instructions can be issued in the same cycle [4]. Migration and the alternate type heuristic are useful in either direction.

busy times for floating point function units - This dynamic heuristic accounts for structural hazards for non-pipelined floating point function units. Earliest execution time can be defined to include this consideration, but, by itself, it is useful in either direction.

maximum path length to a leaf - This is a static heuristic that counts the number of arcs between the candidate node and the most distant leaf. It attempts to balance progress along all paths in the DAG for a forward scheduling pass and must be calculated in a backward-pass manner:

```
foreach ( node in basic block in reverse order ) {
  node->maxpathlength = 0;
  foreach ( child of node ) node->maxpathlength =
     max( node->maxpathlength,
        1 + child->maxpathlength );
}
```

maximum delay to a leaf - This static heuristic is similar to maximum path length to a leaf. It is calculated in a similar manner but may be inaccurate when all transitive arcs are removed.

maximum path length from root - This static heuristic is the backward scheduling pass analogue of maximum path length to a leaf. However, Shieh and Papachristou recommend its use in a forward scheduling pass to help schedule nodes as soon as possible [13]. It must be calculated by a forward pass.

maximum delay from root - This static heuristic is the backward scheduling pass analogue of maximum delay to a leaf. It must be calculated in a forward-pass manner.

earliest start time - Earliest start time is a static heuristic distinct from earliest execution time. The earliest start time of an initial dummy node is zero. Each other node is assigned the maximum of earliest_start(p) + latency(p) over all parents p [12]. This requires a forward pass for calculation and can be part of a critical path algorithm. Note that this measure may be inaccurate when all transitive arcs are removed.

latest start time - The latest start time of a block-terminating dummy node is the value assigned to that node for earliest start time; therefore, this calculation can only begin after the forward pass for earliest start time. Each other node is assigned the minimum of latest_start(c) over all children c then minus the latency of that node [12]. This requires a backward pass for calculation and can be part of a critical path algorithm. Note that this measure may be inaccurate when all

transitive arcs are removed.

slack - Slack is defined as the difference between latest start time and earliest start time; those nodes with a slack of zero are on the critical path.

#children - This is the number of outgoing arcs from a node and is calculated by add_arc, which increments a counter in the node. When used as a heuristic in a forward scheduling pass, #children attempts to estimate how many nodes will be added to the candidate list and thereby provide more flexibility in scheduling choices; however, the true measure is #uncovered children. Note that this measure will be inaccurate for a compare-against-all DAG construction method.

ϕ delays to children - This is a static heuristic similar to #children but for which ϕ=sum includes the delays as weights on the children and thus increases the likelihood that a node with many multi-cycle delay arcs will be scheduled next. For ϕ=max, this heuristic is the same as execution time.

#single-parent children - A single-parent child is a child with only one incoming arc. Thus #single-parent children is a better estimate of how many nodes will be added to the candidate list when a given candidate node is scheduled. Note that transitive arcs cause no problem; however, because of delays greater than one, not all single-parent children necessarily will be added to the candidate list. The true measure is #uncovered children. The calculation requires either that links be deleted from parents upon scheduling, and the corresponding #parents counter be decremented, or the use of a #unscheduled_parents counter. During scheduling, this dynamic heuristic is calculated as follows:

```
#single_parent_children = 0;
foreach ( child of candidate node ) {
 if ( child -> #unscheduled_parents == 1 )
  increment #single_parent_children;
}
 . . .
if ( node scheduled ) {
 foreach ( child of scheduled node )
  decrement ( child -> #unscheduled_parents );
}
```

sum of delays to single-parent children - This is a dynamic heuristic similar to #single-parent children but which includes the delays as weights on the children and thus increases the likelihood that a node with many multi-cycle delay arcs will be scheduled next.

#uncovered children - This heuristic is the refinement of #children and #single-parent children. For a forward scheduling pass it measures exactly how many nodes will be added to the candidate list if a given candidate node is scheduled [16]. Its calculation is the same as that given for #single-parent children above, except that the first if condition is extended to also require that the delay to the child be equal to one. Note that, due to multiple resource definitions and asymmetric bypass paths, #uncovered children can be different from #single-parent children and yet be greater than zero.

#parents - This is the number of incoming arcs to a node, and it will be inaccurate in the presence of transitive arcs. It is calculated by add_arc, which increments a counter in the node. Shieh and Papachristou recommend this as an inverse heuristic for a forward scheduling pass because the larger number of parents will mean that the candidate node must wait for a larger number of instruction completions [13]. Earliest execution time subsumes this heuristic for the forward pass, but #parents might provide an analogue to an uncovering estimate for backward-pass scheduling.

ϕ delays from parents - This is a static heuristic similar to #parents but which for ϕ=sum includes the delays as weights on the parents; thus, for a backward scheduling pass it can increase the likelihood that a node with many multi-cycle delay arcs from its parents will be scheduled next. ϕ=max can also be useful for backward-pass scheduling.

#descendants - This is the total number of nodes connected to a candidate node as children, grandchildren, etc. It is an extension of the idea of uncovering and attempts to provide for more flexibility in forward-pass scheduling. However, it is hard to compute with table-building DAG construction algorithms, and its calculation must avoid double counting when arcs converge on the same descendant node. A node visiting and marking algorithm can be used, but it would be expensive. A better approach is for add_arc to maintain reachability bit maps, whether or not they are used to prevent transitive arcs; the #descendants is then merely the population count on the reachability bit map (i.e., number of set bits) minus one.

sum of execution times of descendants - This is a static heuristic similar to #descendants but which includes weights for the execution times.

#registers born - This is a static measure that can be used as an inverse heuristic on a forward scheduling pass before register allocation, that is, it is more advantageous to postpone scheduling of an instruction that increases the register pressure. This kind of heuristic, and the three following, are useful in prepass scheduling (i.e., before register allocation). In fact, an algorithm like Warren's is designed to be performed both prepass as well as postpass [16]. The integration of register allocation and instruction scheduling into one pass has also been studied by other authors [2,5].

#registers killed - This is a static measure for use in a prepass, forward-pass scheduling algorithm to increase the likelihood of scheduling an instruction that reduces the register pressure. The version 2 GNU C compiler includes this heuristic as a modification to Tiemann's algorithm [17].

liveness - Register liveness is a static measure defined by Warren that includes the concepts of #registers born/killed above; see [16] for a detailed discussion.

birthing instruction - In Tiemann's backward scheduling pass, each RAW parent of the most recently scheduled node has its priority adjusted upward so that each is more likely to be chosen next and thus shorten the lifetime of the corresponding live register [15]. The version 2 GNU C compiler uses a modified Tiemann's algorithm for both prepass and postpass instruction scheduling [17].

4. HEURISTIC CALCULATION STEP

As discussed in the introduction, an intermediate heuristic calculation step may be required as a pass over the DAG to provide any remaining static heuristics left undetermined after DAG construction. Typically a level algorithm is used [8,13]. For forward DAG construction, root nodes are assigned a level of 0; other nodes are assigned the value one plus the maximum level of any parent. A linked list is maintained for each level, for which inserts are performed as part of the add_arc procedure. (An alternate definition of levels for backward-pass DAG construction is to assign leaf nodes a level of zero and other nodes one plus the maximum level of any children.)

A backward intermediate pass can be done by running an outer loop from the maximum level to the minimum. An inner loop visits each node on that level, and an innermost loop visits each child to determine descendant-related information. Thus a parent can examine all its children and know that all descendants have been processed. However, any reverse topological sort, including a reverse scan of the original instructions in the basic block, produces the same result. Thus it is better to construct a linked list of instructions during DAG construction and reverse walk it than constructing a more sophisticated data structure such as an array of level-lists.

5. INSTRUCTION SCHEDULING ALGORITHMS

Table 2 presents an analysis of six published instruction scheduling algorithms, including the DAG construction method and pass, the direction of the scheduling pass, and the ranked usage of heuristics. Gibbons and Muchnick used backward-pass DAG construction to handle condition code dependencies in a special way. Krishnamurthy and Warren have the same DAG construction and scheduling directions and thus require intermediate steps. Some algorithms (e.g., Krishnamurthy) use a postpass "fixup" to try to fill more operation delay slots than are filled by the heuristic scheduling pass.

Some algorithms combine the heuristic information into a single priority value per node, while others apply heuristics in a given order in a winnowing-like process. Of the six, two require the calculation of heuristics in both a forward and backward manner; and, of these two, the requirement is unavoidable in Schlansker. However, the use of minimum path to a root in Shieh and Papachristou could be possibly be omitted or replaced with little effect because it is the last heuristic to be applied. Krishnamurthy provides a survey of several of these algorithms and many others [9].

6. DAG CONSTRUCTION COMPARISONS AND STATISTICS

In this section we compare the three different DAG construction algorithms. In this comparison we emphasize not the particular heuristics nor their order of application, but instead the pairing of DAG construction algorithms with a simple forward scheduling pass. The three algorithms are: n**2 forward (Warren-like); table-building forward (Krishnamurthy-like); and, table-building backward. The following backward static heuristics are used: max path to leaf, max delay to leaf, and max delay to child, Each algorithm makes two passes over the instructions and then one scheduling pass over the DAG. In the first two approaches both passes over the instructions must visit DAG nodes/parents or children. However, the first pass of the third approach merely constructs the linked list and does not have to visit children. Therefore, the third approach eliminates child revisitation overhead and should be more efficient.

Table 3 presents structural data for the C language versions of GNU grep-1.5 grep.c, regex.c, and dfa.c, GNU gcc-1.35 cccp.c (the C preprocessor), and the double precision versions of Linpack and Livermore Loops (all compiled "cc -O4 -S" under SunOS 4.1.1). Also the Fortran-77 versions of tomcatv, nasa7, and fpppp are included (from the SPEC benchmark suite, license #494, compiled "f77 -O4 -S").

Fpppp-1000 has a maximum block size of 1000 instructions, fpppp-2000 has maximum of 2000, and fpppp-4000 has a maximum of 4000. The data in Table 3 are independent of the three approaches. The last column contains the number of unique memory expressions found in load/store instructions; this measures the number of different symbolic memory address expressions found in the SPARC assembly language code. A delay slot instruction, including that for an annulling branch, is included in the counts for the basic block following the branch.

Table 4 gives the running times of the instruction scheduling approach using the n**2 algorithm along with statistics on number of children per instruction and number of arcs per basic block. The timings were collected using /usr/bin/time on a SPARCstation-2 under SunOS

4.1.1 and taking the average of *user* + *sys* over five runs. The tomcatv program is noteworthy because it had fewer total instructions than either linpack or lloops but required longer to schedule; this can be traced to the large number of children per instruction and correspondingly large number of arcs per basic block. Versions of fpppp other than the 1000-instruction maximum were not run for this approach due to the excessive time and space requirements.

Table 5 contains the running times and structural statistics for the two approaches using table-building DAG algorithms. The timings demonstrate that, while the n**2 method is competitive on system codes, the table-building methods are superior as basic block size increases. For the n**2 algorithm to remain practical, an instruction window size (i.e., maximum basic block size) of no more than 300-400 instructions should be maintained (cf. tomcatv and nasa7). The table-building methods do not require the use of instruction windows.

The results in Table 5 also show that the two table-building methods are essentially equivalent even at large basic block sizes. The expected increase in efficiency due to eliminating revisitation overhead in the third approach is insignificant in comparison to the total amount of computation. Indeed, the second approach (forward) requires less time to schedule the full fpppp program than the third approach (backward); upon examination of the source code, this effect is due to the placement of symbolic memory address expressions more toward the end of the large basic block. Thus, the backward-pass DAG construction algorithm encounters these expressions earlier than the forward-pass algorithm and thus has a larger average number of resources for which it must check dependencies (i.e., a variable-length bit map is used to represent resource use and definition and its length is increased whenever a new memory address expression is encountered).

7. CONCLUSIONS

We have investigated several approaches to increasing the efficiency of DAG construction and heuristic calculations for instruction scheduling at the basic block level. Our results have confirmed some of our intuition and disproved other conjectures. Our findings include:

1) The table-building methods are significantly faster for large basic blocks than the compare-against-all (n**2) approach, as expected.

2) The table-building methods are robust and do not require instruction windows even for extremely large basic blocks, which we did not expect.

3) We recommend against the transitive-arc-avoidance improvement for the n**2 approach (i.e., Landskov), since some transitive arcs carry important timing information that would otherwise be unavailable. The table-building methods retain these important arcs.

4) Level algorithms are no better for calculation of remaining static heuristics than a reverse walk of a linked list of the instructions.

5) A systematic survey of 26 heuristics revealed six major categories. For example, the intuitive concept of uncovering has three relationship-based heuristics, of which #uncovered children is the best measure but is harder to calculate than #children.

6) Our conjecture that we should always pair a DAG construction algorithm with an opposite direction scheduling pass was false. Our results showed negligible difference in efficiency for the proposed pairing.

7) Our experiments have provided DAG structural statistics that will be helpful in future research.

We plan to extend this work by determining if an optimal branch-and-bound scheduler would benefit performance for small basic blocks, characterizing the attributes of larger basic blocks that enable certain heuristics to outperform others, and determining the benefits of global scheduling information (e.g., operation latencies inherited from previous basic blocks).

Acknowledgements

The authors wish to thank the anonymous referees and Paul Beusterien of Harris, Lokender Bommisetty and Jeff Camp of Clemson, Jim Dehnert of Silicon Graphics, Andy Glew of Intel, Mary Jean Harrold and Stig Thormodsrud of Clemson, and Jim Wilson of Cygnus for their helpful comments on earlier drafts.

REFERENCES

[1] D. Bernstein and M. Rodeh, "Global instruction scheduling for superscalar machines," *Proc. SIGPLAN Conf. on Prog. Lang. Design and Impl.*, Toronto, June 1991, pp. 241-255.

[2] D.G. Bradlee, S.J. Eggers, and R.R. Henry, "Integrating register allocation and instruction scheduling for RISCs," *Proc. ASPLOS-IV*, Santa Clara, CA, April 1991, pp. 122-131.

[3] P.B. Gibbons and S.S. Muchnick, "Efficient instruction scheduling for a pipelined architecture," *Proc. SIGPLAN Symp. on Compiler Const.*, Palo Alto, CA, July 1986, pp. 11-16.

[4] D.N. Glass, "Compile-time instruction scheduling for superscalar processors," *Proc. COMPCON Spring 90*, San Francisco, CA, 1990, pp. 630-633.

[5] J.R. Goodman and W.-C. Hsu, "Code scheduling and register allocation in large basic blocks," *Proc. Intl. Conf. on Supercomputing*, St. Malo, France, July 1988, pp. 442-452.

[6] J. Hennessy and T. Gross, "Postpass code optimization of pipeline constraints," *ACM Trans. on Prog. Lang. and Systems 5*, 3 (July 1983) 422-448.

[7] D. Hunnicutt, "DDG construction algorithms," M.S. paper, Dept. of Computer Science, Clemson University, May 1991.

[8] S. Krishnamurthy, "Static scheduling of multi-cycle operations for a pipelined RISC processor," M.S. paper, Dept. of Computer Science, Clemson University, May 1990.

[9] S. Krishnamurthy, "A brief survey of papers on scheduling for pipelined processors," *SIGPLAN Notices 25*, 7 (July 1990) 97-106.

[10] D. Landskov, S. Davidson, B. Shriver, and P.W. Mallett, "Local microcode compaction techniques," *ACM Computing Surveys 12*, 3 (September 1980) 261-294.

[11] J.W. Rymarczyk, "Coding guidelines for pipelined processors," *Proc. ASPLOS-I*, Palo Alto, CA, March 1982, pp. 12-19.

[12] M. Schlansker, "Compilation for VLIW and superscalar processors," *ASPLOS-IV Tutorial*, Santa Clara, CA, April 1991, pp. mss-1 - mss-74.

[13] J.J. Shieh and C.A. Papachristou, "On reordering instruction streams for pipelined computers," *Proc. MICRO-22*, Dublin, Ireland, August 1989, pp. 199-206.

[14] R. Sites, "Instruction ordering for the Cray-1 computer," Technical Rept. 78-CS-023, Comp. Sci., University of California, San Diego, July 1978.

[15] M.D. Tiemann, "The GNU instruction scheduler," CS343 course report, Comp. Sci., Stanford University, June 1989.

[16] H.S. Warren, Jr., "Instruction scheduling for the IBM RISC System/6000 processor," *IBM J. Res. and Dev. 34*, 1 (January 1990) 85-92.

[17] J. Wilson, Cygnus Inc., personal correspondence, May 1991.

[18] H. Zima and H. Chapman, *Supercompilers for Parallel and Vector Computers.* New York: ACM Press, 1991.

	relationship-based	timing-based	pass
stall behavior	interlock with previous inst.	earliest execution time **	v
	interlock with child **	execution time	a
inst. class	alternate type	busy times for flt. pt. function units	v
critical path	max path length to a leaf	max total delay to a leaf	b
	max path length from root	max total delay from root	f
	-	earliest start time (EST) **	f
	-	latest start time (LST) **	b
	-	slack (= LST-EST) **	f+b
uncovering	#children **	ϕ delays to children **	a
	#single-parent children	sum of delays to single-parent children	v
	#uncovered children	-	v
structural	#parents **	ϕ delays from parents **	a
	#descendents	sum of execution times of descendents	b
register usage	#registers born	-	a
	#registers killed	-	a
	liveness	-	a
	birthing instruction	-	a

Table 1. Various heuristics

Legend:

- number of
ϕ - maximum or sum function
a - determined when an instruction node or dependency arc is added to DAG
b - requires backward pass over basic block for calculation
f - requires forward pass over basic block for calculation
v - requires node visitation during scheduling pass for calculation
** - calculation is affected by the presence of transitive arcs

	Gibbons & Muchnick [3]	Krishnamurthy [8]	Schlansker [12]	Shieh & Papa-christou [13]	Tiemann (GCC) [15]	Warren [16]
dag construction:						
type of pass	b	f	n.g.	n.g.	f	f
algorithm	n**2	table building	n.g.	n.g.	table building	n**2
scheduling:						
type of pass	f	f+postpass	b	f	b	f
heuristics:		(priority fn)	(priority fn)		(priority fn)	
no interlock w/ previous inst.	1v	-	-	-	-	-
earliest time	-	1v	-	-	-	1v
interlock w/child	2	-	-	-	-	-
execution time	-	4	-	2	-	-
alternate type	-	-	-	-	-	2
fpu interlocks	-	2v	-	-	-	-
max path to leaf	4b	3b	-	-	-	-
max delay to leaf	-	5b	-	1b	-	3b
max path to root	-	-	-	5f	-	-
max delay to root	-	-	-	-	1f	-
slack time	-	-	1f+b	-	-	-
latest start time	-	-	2b	-	-	-
number of children	3	-	-	3	-	-
number uncovered	-	-	-	-	-	5v
number of parents	-	-	-	4	-	-
register liveness	-	-	-	-	-	4
birthing instruction	-	-	-	-	2	-
original order	-	-	-	-	3	6

Table 2. Various scheduling algorithms

Legend:

b - requires backward pass over basic block for calculation
f - requires forward pass over basic block for calculation
v - requires node visitation during scheduling pass for calculation
n.g. - not given in reference
<number> - relative importance of heuristic for this scheduling algorithm

benchmark	# basic blocks	# insts	insts/basic block		unique memory exprs. /basic block	
			max	avg	max	avg
grep	730	1739	34	2.38	5	0.32
regex	873	2417	52	2.77	9	0.31
dfa	1623	4760	45	2.93	13	0.67
cccp	3480	8831	36	2.54	10	0.35
linpack	390	3391	145	8.69	62	2.58
lloops	263	3753	124	14.27	40	4.37
tomcatv	112	1928	326	17.21	68	5.24
nasa7	756	10654	284	14.09	60	4.23
fpppp-1000	675	25545	1000	37.84	120	5.92
fpppp-2000	668	25545	2000	38.24	161	5.34
fpppp-4000	664	25545	4000	38.47	209	5.02
fpppp	662	25545	11750	38.59	324	4.76

Table 3. Structural data for benchmarks independent of approach

benchmark	run time (secs)	children/inst		arcs/basic block	
		max	avg	max	avg
grep	2.2	7	0.70	71	1.66
regex	3.0	8	0.72	107	2.00
dfa	5.3	15	0.89	185	2.61
cccp	8.5	9	0.67	94	1.70
linpack	11.1	34	2.10	1024	18.29
lloops	11.6	22	1.86	651	26.54
tomcatv	16.3	59	4.91	4861	84.53
nasa7	49.4	58	3.62	4659	50.95
fpppp-1000	1522.0	602	55.61	155421	2104.56

Table 4. Scheduling run times and structural data for n**2 approach

benchmark	run times (secs)		children/inst		arcs/basic block	
	forward	backward	max	avg	max	avg
grep	2.0	2.0	4	0.52	42	1.23
regex	2.7	2.7	4	0.53	41	1.46
dfa	4.5	4.5	10	0.62	65	1.81
cccp	8.1	8.0	7	0.52	47	1.31
linpack	3.4	3.4	17	1.02	258	8.88
lloops	3.7	3.7	9	1.07	219	15.29
tomcatv	2.3	2.2	9	1.52	744	26.14
nasa7	9.3	9.2	26	1.26	572	17.73
fpppp-1000	23.2	23.1	185	2.33	3098	88.35
fpppp-2000	23.9	23.6	403	2.43	6345	93.10
fpppp-4000	24.5	24.5	503	2.53	13059	97.15
fpppp	26.5	26.8	503	2.60	37881	100.27

Table 5. Scheduling run times and structural data for table-building approaches