Efficient Instruction Scheduling for a Pipelined Architecture

Phillip B. Gibbons* & Steven S. Muchnick**

Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304-1181

Abstract

As part of an effort to develop an optimizing compiler for a pipelined architecture, a code reorganization algorithm has been developed that significantly reduces the number of run-time pipeline interlocks. In a pass after code generation, the algorithm uses a dag representation to heuristically schedule the instructions in each basic block.

Previous algorithms for reducing pipeline interlocks have had worst-case runtimes of at least $O(n^4)$. By using a dag representation which prevents scheduling deadlocks and a selection method that requires no lookahead, the resulting algorithm reorganizes instructions almost as effectively in practice, while having an $O(n^2)$ worst-case runtime.

1. Introduction

The architecture we have studied has many features which enable fast execution of programs, chief among them the use of pipelining. Whereas in a more traditional architecture each instruction is fetched, decoded and executed before the next one is fetched, in a pipelined architecture [Kog81] the execution cycles of distinct instructions may overlap one another. Problems arise if the results of one instruction are needed by another before the first has finished executing or if a specific machine resource is needed by two instructions at once. In such a case, the second instruction must wait for the first to complete, and we say a *pipeline hazard* has occurred.

*Current Address: Computer Science Division, University of California, Berkeley, CA 94720.

**Current Address: Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, CA 94043.

Fortunately, not all pairs of consecutive instructions cause pipeline hazards. In the architecture under consideration, the only hazards are register- and memory-based: 1) loading a register from memory followed by using *that* register as a source, 2) storing to any memory location followed by loading from any location, and 3) loading from memory followed by using *any* register as the target of an arithmetic/logical instruction or a load/store with address modification. Each of these pipeline hazards causes some potential implementation of the architecture to stall or *interlock* for one pipe cycle.

There are three approaches to reducing the number of pipeline interlocks incurred in executing a program, distinguished by the agent and the time when the code is inspected: either special hardware can do it during execution, or a person or software can do it before execution. The hardware approach has been used in the Control Data 6600 [Tho64] and the IBM 360/91 [Tom67], two of the fastest machines of their day. While reasonably effective, this approach is very expensive and can only span relatively short code sequences. Rymarczyk [Rym82] has presented guidelines for assembly language programmers to avoid pipeline interlocks. This approach is impractical in general, since it is very time-consuming and error-prone. The use of software to detect and remove interlocks is, at present, the most practical and effective approach. Our goal was to design an efficient algorithm for reordering instructions at compile time that significantly reduces the number of interlocks occurring when code is executed on any implementation of the subject architecture.

Most research on compile-time code reorganization has concentrated on the scheduling and compacting of microcode, which begins with a correct sequence of vertical microinstructions and packs them into a (shorter) sequence of horizontal microinstructions. Although pipeline-oriented scheduling is similar to microcode compaction in some respects, there are enough differences that algorithms for it [Dav81, Tok81, Veg82] are not adequate for instruction scheduling. Research on compile-time pipeline scheduling is relatively sparse, with the most significant works being [Ary83], [Aus82], [Gro83], [Hen83] and [Sit78]. [Gro83] presents an excellent and relatively thorough survey of research on this subject. In [Ary83], an algorithm with exponential worst-case runtime is presented for optimally scheduling instructions for a pipelined vector processor. [Aus82] describes pipeline scheduling performed during code generation. The others [Gro83, Hen83, Sit78] (and our own work) are all concerned with scheduling performed during a pass after code generation and register allocation. According to the empirical evidence [Gro83], this can be significantly more effective than the technique of [Aus82]. [Gro83] and [Hen83] describe a powerful heuristic scheduling algorithm that uses lookahead to avoid deadlock, with $O(n^4)$ worst-case runtime, where n is the number of machine instructions in the basic block being scheduled. By using a graph representation which prevents scheduling deadlocks and a selection method that does not require lookahead, we have developed an algorithm which eliminates nearly as many interlocks as theirs while having an $O(n^2)$ worst-case runtime and an observed linear runtime in practice.

2. The Issues

There are two issues which must be addressed when designing an algorithm for instruction scheduling: 1) how to express the constraints which must be satisfied by any legal reordering, and 2) how to determine the order in which instructions are scheduled, subject to the constraints.

Clearly, instructions in a program cannot be reordered arbitrarily. Certain instructions *must* remain ahead of other instructions in the resulting code sequence for the overall effect of the program to remain unchanged. The first issue, then, involves developing a representation that expresses the maximal freedom to rearrange instructions without compromising correctness.

Once the constraints on the reordering have been determined, a method is needed to choose among the possible reorderings, while maintaining reasonable runtimes for the scheduler.

3. The Assumptions

Our architecture has hardware hazard detection and an interlock mechanism, so it is not mandatory that all pipeline hazards be removed. Moreover, the resulting code is intended to run on a range of possible implementations with differing sets of interlocks. This makes it impossible in general to remove all interlocks for all implementations, if one desires to obtain code which will run well on all of them without rescheduling. Thus, our goal was to develop a heuristic algorithm that performs well for all implementations, while perhaps suboptimally for any particular one.

To simplify the task, we made some assumptions as to how the machine code references memory. For instance, while there may be multiple base registers, each memory location is assumed to be referenced via an offset from only one. Furthermore, pointer references are assumed to overlap all memory locations. This preserves correctness in the presence of worst-case aliasing. These assumptions simplify determination of the reordering constraints in practice, but, as will be seen below, do not alter the worst-case runtime bound of $O(n^2)$. Also, the assumptions can be effectively replaced with information about the patterns of memory aliasing obtaining in a program, such as is produced by the compilers described in [Cou86] and [Spi71].

4. The Approach

The overall approach divides the problem into three steps: 1) divide each procedure into basic blocks[1], 2) construct a directed acyclic dependency graph expressing the scheduling constraints within each basic block, and 3) schedule the instructions in the block, guided by the applicable heuristics, using no lookahead in the graph.

Expressing Constraints: The Dependency Dag

We construct for each basic block a directed acyclic graph (dag) whose nodes are the instructions within the block and whose edges represent serialization dependencies between instructions. An edge leading from instruction a to instruction b indicates that a must be executed before b to preserve correctness of the overall program.

An example code sequence (with source operands written before destination operands) and its dependency dag (with roots at the top) are shown in Figs. 1 and 2. For any particular resource, such as a register, the dependency dag serializes definitions vs. definitions (e.g. instruction 5 vs. instruction 9 in Fig. 2), definitions vs. uses (3 vs. 8), and uses vs. definitions (4 vs. 6). The dags take into account *all* serialization constraints, including register dependencies, memory dependencies, processor state-modifying instructions and carry/borrow dependencies[2].

```
1   add     #1,r1,r2
2   add     #12,sp,sp
3   store   r0,A
4   load    -4(sp),r3
5   load    -8(sp),r4
6   add     #8,sp,sp
7   store   r2,0(sp)
8   load    A,r5
9   add     #1,r0,r4
```

Figure 1. Sample code sequence.

In general, the dags are constructed by scanning backward across a basic block, noting each definition or use of a resource and then later the definitions or uses which must precede it. Thus, for example, in Fig. 2 an arrow is inserted between 2 and 4 because instruction 2 defines the stack pointer sp and 4 is the

[1]This step is usually performed by any optimizing compiler and hence is considered to be "free". So as to make basic blocks as long as possible, our compiler does not consider a procedure call or an instruction which conditionally skips a following (non-branch) instruction to end a basic block.

[2]The architecture includes several instructions which set or use carry/borrow bits in the processor state. The collection of these bits is a unique processor resource, like a register, and hence requires serialization. However, as will be seen below, they require special handling in the scheduler.

next instruction in the linear sequence which either uses or defines it[3]. Carry/borrow dependencies are handled specially in constructing the dags, since carries and borrows are very frequently defined but only rarely used. Serializing all carry/borrow definitions against each other would be unduly constraining. Instead, a special subgraph is generated within the dag for each instruction which *uses* a carry or borrow; the subgraph includes all the instructions which must appear between the use and the corresponding definition (or the beginning of the basic block if no definition is found in it). Scanning backward across the instructions of a basic block facilitates this special handling of the carry/borrow bits.

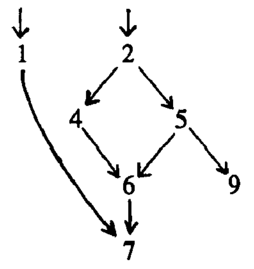

Figure 2. Dependency dag for code in Fig. 1.

Our dags differ from those of [Gro83, Hen83] primarily in that we serialize definitions vs. definitions, while they do not. In the absence of liveness information, this serialization is essential, at least for the final definition of a resource in a basic block. It also avoids the possibility of deadlocks in the scheduling algorithm.

Selecting an Order: The Static Evaluator

As long as the instructions in a basic block are scheduled in some topological sort [Knu68] of the dependency dag, the overall effect[4] of the block is indistinguishable from its execution in the original order.

Our algorithm selects instructions to schedule by sweeping down the dag, beginning at the roots (which represent the instructions which can possibly be executed first). An instruction is a *candidate* for scheduling if all its immediate predecessors in the dag have been scheduled (or if it has no predecessors). Among the candidates at any given time, the "best" instruction is selected based on the following two guidelines: 1) if possible, an instruction is scheduled that will not interlock with the one just scheduled, and 2) given a choice, an instruction is scheduled that is most likely to cause interlocks with instructions after it. The first guideline is obvious: it is the local expression of the overall goal of the scheduling process. When the second is combined with it, the selections can be thought of as occurring in pairs comprising an instruction likely to cause an interlock, followed by one which does not interlock with it (but which is likely to cause an interlock itself). In our architecture, for example, the best choice after an "add" is a "load", while the best choice after a "store" is another "store". These heuristics are incorporated into a *static* evaluator which rates individual candidates.

Looking ahead in a basic block to instructions which are not yet candidates will certainly improve scheduling. Unfortunately, this lookahead dramatically increases worst-case runtime. Instead, three heuristics[5] are used in place of lookahead. The heuristics express several static local properties of nodes in a scheduling dag. In order of importance to the scheduling process, they are:

1) whether an instruction interlocks with any of its immediate successors in the dag,
2) the number of immediate successors of the instruction, and
3) the length of the longest path from the instruction to the leaves of the dag.

Intuitively, these properties bias toward selecting instructions which:

1) may cause interlocks (and hence need to be scheduled as early as possible, when there is most likely to be a wide choice of instructions to follow them),
2) uncover the most potential successors (and hence the widest possible latitude for future choices), and
3) balance the progress along the various paths toward the leaves of the dag (and hence leave the largest number of choices available at all stages of the process).

An outline of the scheduling algorithm, given a basic block of machine instructions, is as follows:

1. make a prepass backward over the basic block to construct the scheduling dag, comparing each instruction to the nodes of the scheduling dag constructed so far[6]
2. put the roots of the dag into the candidate set (an instruction is a *root* if it has no predecessors in the dag)
3. select the first instruction to be scheduled from the candidate set, taking into account the instructions which terminate its predecessor basic blocks and the static heuristics (applied in the order given in the preceding paragraph)

[3]A minimal set of edges suffices to represent the dags. The full set of dependencies is the transitive closure of the binary relation given by the edges.

[4]Note the emphasis on "overall effect". Since we are rearranging instructions over spans of several statements, there may not be any points in the resulting instruction sequence corresponding to the ends of source statements internal to the block. This may impact the user's understanding of the state of a program whose execution is interrupted, either by the occurrence of a fault or during the debugging process. The interaction of debugging and optimizing transformations has been considered in [Hen81, Zel84].

[5]For a discussion of these and similar heuristics and their effects on scheduling algorithms in general, see Section 6-8 of [Con67].

[6]Recall that the roots of the completed dag represent the instructions which may appear as the first instruction in the reordered block, so that construction of the dag (sweeping backward across the block) selects the leaves first.

4. while the candidate set is nonempty:
   a. evaluate the candidates based on the last instruction scheduled and the static heuristics (applied in the order given in the preceding paragraph) and select the best one
   b. emit the selected instruction
   c. delete the newly scheduled instruction from the candidate set and add any newly exposed candidates to it

Running our algorithm on the code in Fig. 1 results in the schedule 3, 2, 4, 5, 8, 1, 6, 7, 9 shown in Fig. 3, which reduces the four interlocks in the original sequence (3 - 4, 5 - 6, 7 - 8 and 8 - 9) to one (8 - 1).

```
3   store   r0,A
2   add     #12,sp,sp
4   load    -4(sp),r3
5   load    -8(sp),r4
8   load    A,r5
1   add     #1,r1,r2
6   add     #8,sp,sp
7   store   r2,0(sp)
9   add     #1,r0,r4
```

Figure 3. Result of Scheduling the Instruction Sequence in Fig. 1.

5. Computational Complexity of Our Algorithm

The complexity of our instruction scheduler is at worst $O(n^2)$ for a basic block of n instructions. To build the dependency dag, in the worst case each instruction must be compared with all the instructions already in the dag. Thus building the dag is at most $O(n^2)$. To schedule the instructions, in the worst case all unscheduled instructions must be visited each time an instruction is scheduled. Since evaluating a candidate for scheduling is done without lookahead, the visitation time is $O(1)$, and thus scheduling is $O(n^2) * O(1) = O(n^2)$. Actual scheduling time is generally linear in practice; for example, just over *two* evaluations per instruction are made in scheduling the Acker benchmark [Hen83].

Hennessy and Gross [Gro83, Hen83] present an instruction scheduling algorithm with $O(n^4)$ worst-case runtime. The architecture for which they derived their algorithm imposes more stringent requirements than ours: the hardware has no pipeline hazard detection and no interlock mechanism, and the duration of a pipeline hazard can be more than one cycle. Under our easier requirements, however, their algorithm is still $O(n^4)$. Moreover, the algorithm presented here can be modified to remove all hazards, and hence be effective for their architecture: simply insert a "no-op" whenever a hazard is unavoidable. The algorithm can also be extended to handle multi-cycle pipeline hazards. However, this changes its complexity to $O(b) * O(n^2)$, where b is the upper bound on the duration of a hazard, presumably a constant.

The algorithm presented here is simpler, partly due to the differences between the dags used. As mentioned above, our dependency dags are more restrictive than those in [Gro83, Hen83] in that ours serialize multiple definitions of the same register (while theirs do not) and consider memory and other resources, and hence are somewhat less versatile for scheduling. However, our dependency dags prevent any scheduling deadlocks: there is always an instruction that can be scheduled, regardless of which instructions have been scheduled already. Thus the algorithm does not need lookahead to avoid deadlock. Since the architecture has 32 general registers and the register allocator uses different registers for different temporary values as much as possible[7], serializing definitions does not unduly restrict our code.

6. Experience

We implemented the instruction scheduler described above in C, adding to it a branch scheduler and a floating-point scheduler. The branch scheduler attempts to fill delay slots following branches with instructions selected from the preceding basic block or from the target basic blocks. Our experience with sample code scheduled by it yields the following observations:

1) In general the algorithm performs quite well, despite the comparatively restrictive dags and the lack of lookahead. Without inserting "no-ops", it removes 15 of the 19 avoidable interlocks in the code our C compiler generates for the Acker benchmark. Moreover, for many of the proposed machine implementations, it removes 100% of the avoidable interlocks. On the other end of the scale, it removes only 5 of the 16 avoidable interlocks in the Sieve benchmark.

2) Our assumptions about memory referencing greatly improve the overall results. For example, without our assumption that a single memory location must be referenced using only a single base register, the algorithm removes only 9 of the 19 avoidable interlocks in Acker. On the other hand, *better* aliasing information greatly improves the effectiveness of the algorithm on certain programs. With better aliasing information (as discussed in Section 3 above), it can remove 10 of the 16 avoidable interlocks in Sieve.

3) The carry/borrow subgraphs do not significantly improve scheduling for most programs. Substantial improvements come only for programs which are computationally intensive. The improvements in these cases, however, seem to warrant the small additional cost of constructing the subgraphs.

4) Using the more versatile dags of [Gro83, Hen83] marginally improves the effectiveness of the instruction scheduler on our architecture. Using the same

[7][Gro83, pp. 62 - 63] shows a real-life example where register reuse policy makes a dramatic difference in scheduling.

assumptions about memory referencing and the same heuristics for selection when a choice is available among instructions, our algorithm and that given in [Gro83, Hen83], in fact, produce identical results on the Acker, Sieve and Fibonacci benchmarks.

Additional information on the performance of this algorithm can be found in the paper [Joh86] in this proceedings.

7. Conclusions

We have presented a highly efficient instruction scheduling algorithm for a pipelined architecture which demonstrates the effectiveness of judiciously chosen heuristics and the balancing of policies in other parts of the compilation process (e.g. the register reuse policy) with the approach to scheduling.

Another benefit of this approach to instruction scheduling is that our dependency dags are useful for many other code optimizations. For example, candidates for code hoisting and loop-invariant code motion can readily be discovered using them. Also, a peephole optimizer based on the dags can outperform one based on linear code sequences: the dags expose more combinations of instructions that can be folded, since they tend to bring related instructions closer together.

One issue we have not pursued is the extension of instruction scheduling across basic blocks, which is of particular interest for architectures with long pipelines and either multiple execution units or branch prediction (or both). Techniques for extended basic blocks would seem to be a relatively straightforward extension. The trace scheduling techniques of [Fis81] are relevant as they function, in effect, to lengthen basic blocks.

Another area of interest would be to develop a model of an average instruction sequence and from it an expected runtime for the algorithm.

Acknowledgements

The work described in this paper was conducted during development of an early prototype optimizer for the HP Precision Architecture. The work has been extended, modified and tuned at Hewlett-Packard during the past two years as part of the ongoing development of a product-quality optimizer. See [Joh86] for more recent information and additional performance data.

We thank Carol Hammond, Alexand Wu and Robert Ballance for their contributions to the development and implementation of the scheduling algorithm and James Gosling, John Hennessy, Brian Reid and Lawrence Weber for their helpful comments on an earlier version of this paper.

References

[Ary83] Arya, S. *Optimal Instruction Scheduling for a Class of Vector Processors: An Integer Programming Approach.* Tech. Rept. CRL-TR-19-83, Computer Research Laboratory, the Univ. of Michigan, Ann Arbor, April 1983.

[Aus82] Auslander, M. & M. Hopkins. An Overview of the PL.8 Compiler. *Proc. ACM SIGPLAN Symp. on Compiler Construction,* Boston, June 1982, pp. 22 - 31.

[Con67] Conway, R.W., W.L. Maxwell & L.W. Miller, **Theory of Scheduling,** Addison-Wesley, Reading, MA, 1967.

[Cou86] Coutant, D.S. Retargetable High-Level Alias Analysis, *Proc. ACM Symp. on Princ. of Prog. Lang.*, St. Petersburg Beach, FL, January 1986, pp. 110 - 118.

[Dav81] Davidson, S., D. Landskov, B.D. Shriver & P.W. Mallett. Some Experiments in Local Microcode Compaction for Horizontal Machines. *IEEE Trans. on Computers,* Vol. C-30, No. 7, July 1981, pp. 460 - 477.

[Fis81] Fisher, J.A. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Trans. on Computers,* Vol. C-30, No. 7, July 1981, pp. 478 - 490.

[Gro83] Gross, T.R. *Code Optimization of Pipeline Constraints.* Tech. Rept. 83-255, Computer Systems Lab., Stanford Univ., Dec. 1983.

[Hen81] Hennessy, J.L. Symbolic Debugging of Optimized Code, *ACM Trans. on Prog. Lang. and Sys.*, Vol. 3, No. 1, Jan. 1981, pp. 200 - 206.

[Hen83] Hennessy, J.L. & T.R. Gross. Postpass Code Optimization of Pipeline Constraints. *ACM Trans. on Prog. Lang. and Sys,* Vol. 5, No. 3, July 1983, pp. 422 - 448.

[Joh86] Johnson, M.S. & T.C. Miller. Effectiveness of a Machine-Level, Global Optimizer, *Proc. of the SIGPLAN '86 Conf. on Comp. Constr.*, June 1986.

[Knu68] Knuth, D.E. **Fundamental Algorithms**, Addison-Wesley, Reading, MA, p. 258.

[Kog81] Kogge, P.M. **The Architecture of Pipelined Computers**, McGraw-Hill, New York, 1981.

[Rym82] Rymarczyk, J.W. Coding Guidelines for Pipelined Processors, *Proc. of the Symp. on Arch. Supt. for Prog. Lang. and Oper. Syst.*, Palo Alto, CA, March 1982, pp. 12 - 19.

[Sit78] Sites, R.L. *Instruction Ordering for the Cray-1 Computer.* Tech. Rept. 78-CS-023, Univ. of California, San Diego, July 1978.

[Spi71] Spillman, Thomas C., Exposing Side-Effects in a PL/I Optimizing Compiler, *Information Processing 81*, North-Holland, 1972, pp. 376 - 381.

[Tho64] Thornton, J.E. Parallel Operation in the Control Data 6600, *Proc. Fall Joint Comp. Conf.*, Part 2, Vol. 26, 1964, pp. 33 - 40.

[Tok81] Tokoru, M., E. Tamura & T. Takizuka. Optimization of Microprograms. *IEEE Trans. on Computers*, Vol. C-30, No. 7, July 1981, pp. 491 - 504.

[Tom67] Tomasulo, R.M. An Efficient Algorithm for Exploiting Multiple Arithmetic Units, *IBM J. of Res. and Devt.*, Vol. 11, No. 1, Jan. 1967, pp. 25 - 33.

[Veg82] Vegdahl, S. *Local Code Generation and Compaction in Optimizing Microcode Compilers*, Ph.D. thesis, Carnegie-Mellon Univ., Dec. 1982.

[Zel84] Zellweger, P.T. *Interactive Source-Level Debugging of Optimized Programs*, Research Report CSL-84-5, Xerox Palo Alto Research Center, Palo Alto, CA, May 1984.