Instruction scheduling beyond basic blocks

by M. C. Golumbic
V. Rainish

Instruction scheduling consists of the rearrangement or transformation of program statements, usually at the intermediate language or assembly code level, in order to reduce possible run-time delays between instructions. Such transformations must preserve data dependency and are subject to other constraints. Highly optimizing compilers employing instruction-scheduling techniques have proven to be effective in improving the performance of pipeline processors. Considerable attention has been given to scheduling code within the scope of basic blocks, i.e., straight-line sections of code. In this paper we present techniques for scheduling beyond basic blocks. This allows a further reduction in run-time delays such as those due, e.g., to branches and loops, enabling the exploiting of pipeline architectures which would not otherwise be possible.

Introduction

Instruction-scheduling algorithms are used in compilers to reduce run-time delays for the compiled code. This can be particularly advantageous when compiling for pipeline machine architectures, which allow increased throughput by overlapping instruction execution. For example, if there is a delay of one cycle between fetching and using a value V, it would be desirable to "cover" this delay with an instruction that is independent of V and is "ready" to be executed. Thus, a simple reordering of selected instructions at compile time can be regarded as a form of exploiting the potential parallelism inherent in the code.

Recent research on instruction scheduling for pipeline machines has focused primarily on optimizations within basic blocks [1–6]. A basic block (BB) scheduler generally attempts to interleave independent instructions within each basic block so as to eliminate wasted machine cycles. Such schedulers are quite effective for programs with long basic blocks, common in some scientific applications. Branch instructions, however, restrict the effectiveness of the pipeline architecture in ways that cannot be handled with only basic block transformations.

We have therefore investigated techniques for scheduling beyond the scope of basic blocks. We assume that a good BB scheduler (like that in [5]) is available, and we develop further transformations to help eliminate

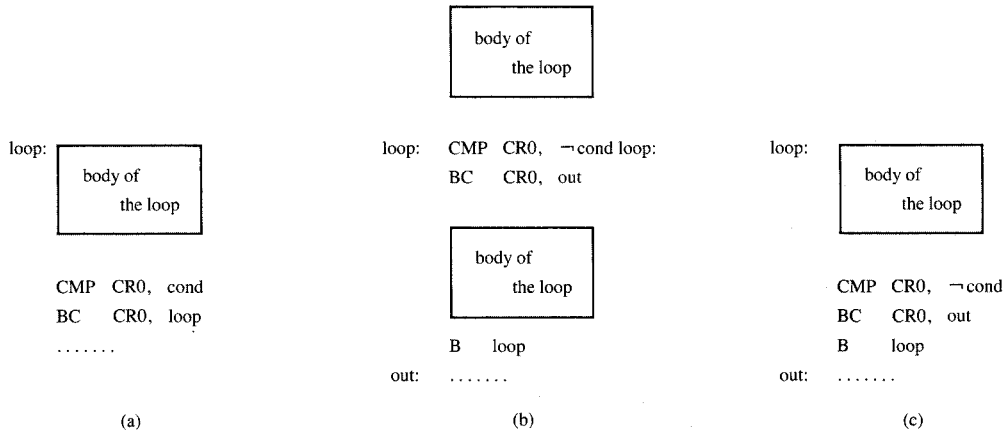

Figure 1

Possible schemes for covering the CMP–BC pair with a BB scheduler.

remaining delays (no-ops) that result from branches and loops. The effect of the delay between a compare and a subsequent conditional branch, which occurs only if the branch is taken, suggests a variety of different styles of loop layout and code replication. These are presented in the next section. We then turn our attention to transformations for covering the delays associated with *if-then-else* statements.

Loop transformations for reducing delays in the IBM RISC System/6000* processor

The *compare–conditional branch* (CMP–BC) pair produces a three-cycle delay along the target path (i.e., if the branch is taken). There is no delay along the sequential path (i.e., if the branch is not taken). The strategy of a scheduler is to try to eliminate this delay (partly or completely) by putting one or more instructions between the compare (CMP) and the conditional branch (BC) to "cover" the delay. If the total time t of performing these instructions is three cycles or more, no delay occurs along any path (target or sequential). However, when t is less than three cycles, a delay of $(3 - t)$ cycles occurs if the branch is taken. In a similar fashion, the instruction triple *compare–conditional branch–unconditional branch* (CMP–BC–B) produces a four-cycle delay if the branch is *not* taken.

In most compilers, including the PL.8 compiler [7], the *exit loop* condition is checked at the end of the loop, as in scheme (a) of **Figure 1**. If the CMP–BC pair is only partly covered by a BB scheduler, the delay will occur for each iteration of the loop. An easy way to eliminate this delay might be to generate loops according to scheme (b)—to check the negation of the original exit loop condition at the beginning of the loop and add an unconditional branch (B) at the end of the body. In such a case a delay would occur only upon exiting the loop, at the expense of replicating the entire body. However, because of the CMP–BC–B triple, it may now be necessary to cover the CMP–BC pair with instructions moved up from the body of the loop; a BB scheduler, whose scope is limited to a single basic block, will not be able to cover (even partly) the CMP–BC, since it is itself a two-statement basic block. Similarly, generating loops as in scheme (c), to favor zero delay for the sequential path of the CMP–BC pair, requires covering instead the CMP–BC–B triple because of the conditional branch followed by the unconditional branch.

Our approach assumes that we have a sophisticated BB scheduler which accepts loops generated according to scheme (a) and succeeds in covering the CMP–BC pair as much as possible within the scope of the basic block. Let scheme (a′) in **Figure 2** denote the loop after BB scheduling has attempted to cover the delay. If the CMP–BC pair is only partly covered ($t < 3$), then perform the following loop-code replication transformation:

1. Create new label (out) before first command following the loop (if one does not already exist).
2. Copy the first $4 - t$ instructions from the beginning of the loop after the BC. (If the number n of instructions

* RISC System/6000 is a trademark of International Business Machines Corporation.

in the beginning of the loop is less than $4 - t$, then copy only those n instructions.)

3. Create new label (loop1) after instruction number $(4 - t)$.
4. Negate the condition of the CMP, change the target of the BC, and add an unconditional branch B following the copied instructions, as illustrated in scheme (d) of Figure 2.

Solely from the point of view of global register allocation [8, 9], such a transformation could be performed either before or after register allocation; it does not influence the outcome of global coloring,[†] nor does coloring restrict the transformation. However, because of coalescing and certain other optimizations [10], it is preferable to perform it after register allocation.

After this transformation is executed, the label "loop:" may become an unused label and, if so, be eliminated; in addition, "straightening" may merge some of the basic blocks. At that point, constant propagation and BB scheduling can be run again on such merged blocks to obtain further improvement.

In general, our transformation, illustrated for loops, could be applied similarly to any conditional branch for which the likelihood of the target path is qualitatively estimated to be greater than the likelihood of the sequential path. One might obtain such qualitative information from source-code analysis, running-test data, or *a priori* knowledge.

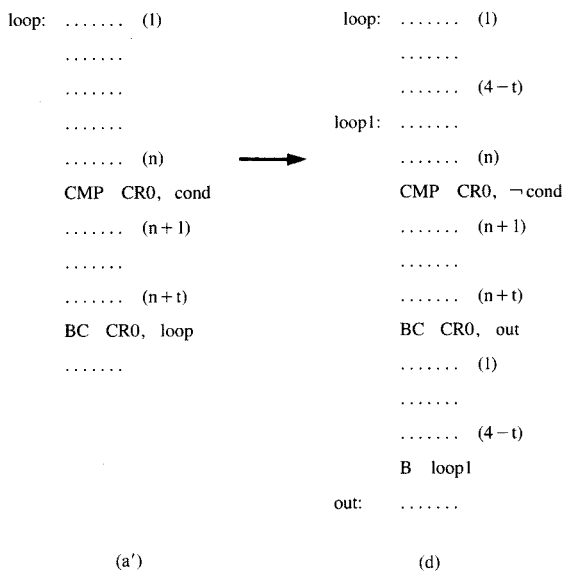

Figure 2

Code-replication transformation to cover a CMP–BC pair that is otherwise only partly covered by BB scheduling.

If-then-else statements with short *then* and long *else*

The CMP–BC–B triple, which produces a four-cycle delay if the branch is not taken, appears frequently with instructions generated for *if-then-else* statements with a "short *then*" part; an example is shown in **Figure 3**. In this sequence of instructions, a three-cycle delay occurs if the conditional branch is not taken, since the CMP–BC–B triple is covered by one instruction only (ADD a,a,1); a three-cycle delay also occurs if the conditional branch is taken (assuming that the "out" block is long enough, say three to four instructions).

It is possible to recognize such sequences in the intermediate language code and transform them by reversing the roles of *then* and *else*, as shown in **Figure 4**. This transformation can easily be applied after register assignment. By waiting until after register allocation, the transformation has a more realistic code sequence to analyze; that is, the assessment of whether the *then* clause is shorter than the *else* clause will be accurate.

Table 1 shows the difference in the instruction delays in our example before and after the transformation;

Source statements		Generated instructions	
if (condition) *then*		CMP	CR0, cond
do;		BC	CR0, else
a = a+1;		ADD	a,a,1
end;		B	out
else	else:	ADD	a,a,b
do;		ADD	a,a,c
a = a+b+c+d+e;		ADD	a,a,d
end;		ADD	a,a,e
out:	out:	.....	
.....		.....	

Figure 3

An *if-then-else* sequence with a "short *then*" part.

clearly, the transformation is always worthwhile. The next section, however, presents a more general technique which includes this optimization.

[†] This code-replication transform may lengthen the "live" area of some variables, but it does not introduce any new edges in the interference graph.

	Before transformation		After transformation
	CMP CR0, cond		CMP CR0, ¬cond
	BC CR0, else		BC CR0, then
then:	.....	else:	.....
	B out		.....
else:	.....		.....
	.....		.....
	.....		B out
	.....	then:	.....
out:	.....	out:	.....
	.....		.....

Figure 4

Transformation of the sequence in Figure 3.

	Before gluing		After gluing
	CMP CR0, cond		CMP CR0, cond
	BC else		BC new_else
then:	instruction_then	new_then:	instruction_then
	B out		instruction_out_1
else:	instruction_else		.....
out:	instruction_out_1		.....
	.....		.....
	.....		instruction_out_n
	.....		B out1
	.....	new_else:	instruction_else
end_out:	instruction_out_n		instruction_out_1
out1:	.....		.....
			.....
			.....
			instruction_out_n
			B out1
		out:	instruction_out_1
			.....
			.....
			instruction_out_n
		out1:	.....

Figure 5

Example of a "gluing" transformation.

Table 1

	Then	*Else*
Delay before transformation	3	3
Delay after transformation	3	0

Glued blocks for speedup of *if-then-else* statements

For statements with both "short *then*" and "short *else*" parts, the transformation described in the preceding section is not applicable. For example,

if [$a(i) > 0$]
  then count1 = count1 + 1;
else count2 = count2 + 1;

produces a three-cycle delay if $a(i) > 0$ because of the CMP–BC–B triple, and a three-cycle delay due to the CMP–BC pair otherwise. In certain situations, which we now describe, we may reduce the delay.

In this transformation, which we call *gluing* (see **Figure 5**), we copy the basic block following the *if* statement and create two new basic blocks, *new_then* and *new_else*. After such a transformation, the delay along the target path remains the same as before, but the delay along the sequential path is reduced (and probably disappears).

If the *out* block ends with an unconditional branch, we can copy it without any change, but if it ends with an instruction which is not a branch, we must add an unconditional branch to the end of the blocks of both the *new_then* and *new_else*.

In these two cases we can create new blocks before the first call to the scheduler, because "gluing" may help to cover register-holding delays in the *out* block itself. If the *out* block ends with CMP–BC, we can copy only the instructions before the BC, create a new label before the BC, and put an unconditional branch to this label at the end of the *new_then* and *new_else* blocks.

In general, gluing can be applied in a similar manner whenever we find a CMP–BC–B triple that is covered by fewer than four instructions. The transformation increases the size of basic blocks and creates new opportunities for constant propagation.

Summary

To take greatest advantage of pipelined architectural features, we have presented a number of structural transformations of loops and branches, thus extending the "vision" of a code scheduler beyond basic blocks. These transformations augment a basic block scheduler that is already good, and extend its capability to cover delays.

The techniques presented here have been implemented in the context of a future version of the PL.8 compiler. Experiments indicate an average run-time speedup of four to five percent beyond that achieved by the BB scheduler [5] on code compiled for the RISC System/6000 CPU.

Acknowledgments

The authors would like to express their thanks to D. Bernstein, M. E. Hopkins, P. W. Markstein, R. Pinter, and H. S. Warren, Jr. for many fruitful discussions.

References

1. D. Bernstein, H. Boral, and R. Y. Pinter, "Optimal Chaining in Expression Trees," *IEEE Trans. Computers* **37,** No. 11, 1366–1374 (November 1988).
2. P. B. Gibbons and S. S. Muchnick, "Efficient Instruction Scheduling for a Pipelined Architecture," *Proceedings of the ACM Symposium on Compiler Construction*, ACM Press, New York, June 1986, pp. 11–16.
3. J. R. Goodman and W.-C. Hsu, "Code Scheduling and Register Allocation in Large Basic Blocks," *Proceedings of the International Conference on Supercomputing*, St. Malo, France, ACM Press, New York, July 1988, pp. 442–452.
4. J. L. Hennessy and T. Gross, "Postpass Code Optimization of Pipeline Constraints," *ACM Trans. Prog. Lang. & Syst.* **5,** 422–448 (July 1984).
5. H. S. Warren, Jr., "Instruction Scheduling for the IBM RISC System/6000 Processor," *IBM J. Res. Develop.* **34,** 85–92 (1990, this issue).
6. S. Weiss and J. E. Smith, "A Study of Scalar Compilation Techniques for Pipelined Supercomputers," *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, CA, October 1987; IEEE, New York, 1987, Catalog No. 87CH2440-6, pp. 105–109.
7. M. A. Auslander and M. E. Hopkins, "An Overview of the PL.8 Compiler," *Proceedings of the ACM Symposium on Compiler Construction*, ACM Press, New York, June 1982, pp. 22–31.
8. G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register Allocation via Coloring," *Comput. Lang.* **6,** 47–57 (1981).
9. G. J. Chaitin, "Register Allocation and Spilling via Graph Coloring," *Proceedings of the ACM Symposium on Compiler Construction*, ACM Press, New York, June 1982, pp. 98–105.
10. D. Bernstein, D. Q. Goldin, M. C. Golumbic, H. Krawczyk, Y. Mansour, I. Nahshon, and R. Y. Pinter, "Spill Code Minimization Techniques for Optimizing Compilers," *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, ACM Press, New York, June 1989, pp. 258–263.

Received February 28, 1989; accepted for publication January 30, 1990

Martin Charles Golumbic *IBM Israel Science and Technology Center, Technion City, Haifa, Israel.* Dr. Golumbic is a research staff member at the IBM Israel Science and Technology Center and an associate professor at Bar-Ilan University. He is the founding editor-in-chief of the series *Annals of Mathematics and Artificial Intelligence*, a member of the editorial board of the journal *Discrete Applied Mathematics*, and a member of the advisory board of the *International Journal of Expert Systems: Research and Applications*. Dr. Golumbic received his Ph.D. in mathematics from Columbia University in 1975. Before moving to Israel in 1982, he served as an assistant professor of computer science at the Courant Institute of Mathematical Sciences of New York University and a visiting scientist at the Université de Paris and the Weizmann Institute of Science. He is the author of the book *Algorithmic Graph Theory and Perfect Graphs* and many research articles in the areas of combinatorial mathematics, algorithmic analysis, expert systems, artificial intelligence, and programming languages. Dr. Golumbic has been a guest editor of *Discrete Mathematics*, associate editor of the volume "Approaches to Intelligent Decision Support," *Annals of Operations Research 12*, and editor of the forthcoming book *Advances in Artificial Intelligence, Natural Language and Knowledge-based Systems*. His current area of research is in combinatorial mathematics interacting with real-world problems in computer sciences and artificial intelligence. He is a member of Phi Beta Kappa, Pi Mu Epsilon, Phi Kappa Phi, and Phi Eta Sigma.

Vladimir Rainish *IBM Israel Science and Technology Center, Technion City, Haifa, Israel.* Mr. Rainish has been a research fellow at the IBM Israel Scientific Center since 1988. He received the M.Sc. and B.Sc. degrees in computer science from the Moscow Transport Institute in 1981. After moving to Israel in 1985, he worked at Scitex Corporation, Herzliya. Mr. Rainish's current area of research is code optimization for pipelined machines.