Instruction Scheduling Across Control Flow

MARTIN CHARLES GOLUMBIC* AND VLADIMIR RAINISH

IBM Israel Science and Technology, MATAM-Advanced Technology Center, Haifa 31905, Israel

ABSTRACT

Instruction scheduling algorithms are used in compilers to reduce run-time delays for the compiled code by the reordering or transformation of program statements, usually at the intermediate language or assembly code level. Considerable research has been carried out on scheduling code within the scope of basic blocks, i.e., straight line sections of code, and very effective basic block schedulers are now included in most modern compilers and especially for pipeline processors. In previous work Golumbic and Rainish: IBM J. Res. Dev., vol. 34, pp. 93–97, 1990, we presented code replication techniques for scheduling beyond the scope of basic blocks that provide reasonable improvements of running time of the compiled code, but which still leaves room for further improvement. In this article we present a new method for scheduling beyond basic blocks called *SHACOOF*. This new technique takes advantage of a conventional, high quality basic block scheduler by first suppressing selected subsequences of instructions and then scheduling the modified sequence of instructions using the basic block scheduler. A candidate subsequence for suppression can be found by identifying a region of a program control flow graph, called an S-region, which has a unique entry and a unique exit and meets predetermined criteria. This enables scheduling of a sequence of instructions beyond basic block boundaries, with only minimal changes to an existing compiler, by identifying beneficial opportunities to cover delays that would otherwise have been beyond its scope. © 1994 by John Wiley & Sons, Inc.

1 INTRODUCTION

Instruction scheduling is a process of rearranging or transforming program statements before execution by a processor in order to reduce possible run-time delays between compiled instructions. An instruction scheduler is normally implemented as part of a compiler [1], and usually operates at an intermediate language (IL) or assembly code level. Such transformations must preserve data dependency and are subject to other constraints. Instruction scheduling can be particularly advantageous when compiling for pipelined 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.

Previous work on the implementation of instruction scheduling concentrated on scheduling within basic blocks [2–7]. A basic block is a sequence of consecutive instructions for which the flow of control enters at the beginning of the sequence and exits at the end thereof without a branch possibility, except at the point of exit. A basic block scheduler attempts to interleave independent instructions within each basic block so as

Received February 1992
Accepted May 1993

*Current address: Dept. of Mathematics and Computer Science, Bar-Ilan University, Ramat Gan, Israel

© 1994 by John Wiley & Sons, Inc.

Scientific Programming, Vol. 2, pp. 1–5 (1993)
CCC 1058-9244/94/030001-05

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 pipelined architecture in ways that cannot be handled with only basic block transformations.

In an earlier work [8], we have investigated *code replication techniques* for scheduling beyond the scope of basic blocks, resulting in reasonable improvements of running time of the compiled code. However, the approach described there still leaves room for further improvement, and is unrelated to the new method presented here.

In this paper, we present a technique called SHACOOF for **S**c**H**eduling **A**cross **CO**ntr**O**l **F**low, which extends capabilities well beyond basic blocks. This technique enables reductions in runtime delays, due to branches and loops, etc., and enables pipelined architectures to be exploited in ways that would not otherwise be possible. The method depends on a new, but simple, decomposition in which successively larger portions of the program control flow graph are replaced by summary pseudo instructions, resulting in new, larger sections of straight line code that can be scheduled by existing techniques.

In Section 2, we introduce the notion of an S-region, and describe in Section 3 how S-regions are used to identify subsequences of instructions for suppression. A detailed example showing how the SHACOOF instruction scheduler covers delays in a simple yet typical program is given in Section 4 along with a discussion of implementation issues.

2 S-REGIONS

The fundamental idea of the SHACOOF method is to identify a candidate subsequence of instructions to be suppressed, treating it as one would a subroutine, making it transparent to the basic block scheduler. These subsequences correspond to regions of the program flow graph having a unique entry, a unique exit, and satisfying certain minimality conditions. We call these *S-regions.* By suppressing the S-region, regarding it as a pseudo instruction, and preserving data dependency, instructions can then be moved over it using the current basic block scheduler. We now make these definitions precise.

Let G be the program *flow graph,* with vertices corresponding to the basic blocks and with a directed edge from block B_1 to B_2 if B_2 is a successor of B_1 [1]. Let $pred(N)$ and $succ(N)$ denote the set of all predecessors and successors of vertices of set N, respectively. A subgraph S of G is called an *S-region* with entry x and exit $y (x \neq y)$ if the following four conditions hold:

1. $x, y \in S$
2. $pred(S - x) \subseteq S - y$
3. $succ(S - y) \subseteq S - x$
4. There is no region $S' \subseteq S$, with entry x' and exit y' satisfying (1) − (3), with $x = x'$ or $y = y'$.

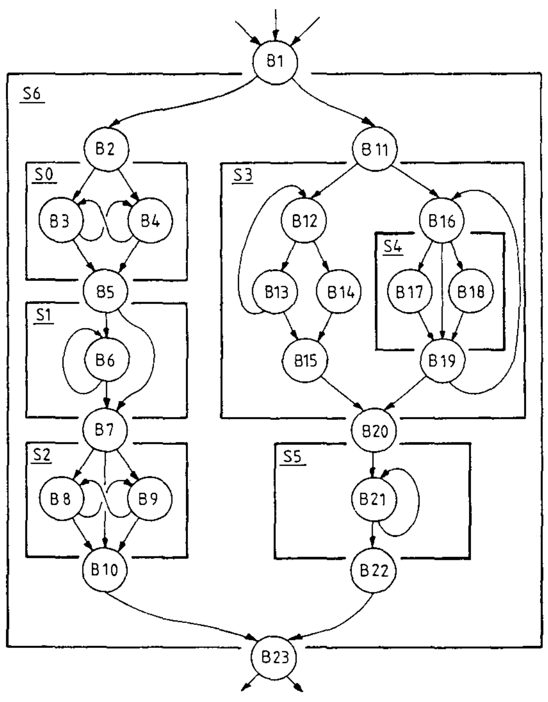
In particular, the definition implies that every path from $G - S$ to S goes through x, and that every path from S to $G - S$ goes through y. It is also easy to show the following.

Lemma. There is at most one S-region with entry x or with exit y.

Proof. Suppose there are two S-regions S_1 and S_2 with the same entry x. Condition 4 implies that one is not contained in the other, so there exists a $z \in S_1$ $(z \neq x)$, such that $z \notin S_2$. But there must be a path from x to z, so $z \in succ(S_2 - y_2)$, where y_2 is the exit of S_2. Condition 3 implies that $z \in S_2 - x$, a contradiction. □

S-regions may be nested or chained as illustrated in Figure 1. This example shows part of a flow graph, where each vertex B_1 to B_{23} represents a basic block and $S_0, \ldots, S_6$ denote S-regions. In the example, $S_0 - S_1 - S_2$ and $S_3 - S_5$ are chained S-regions; region S_4 is nested inside region S_3, etc. S-regions can be generated or recognized in $O(e \log e)$ time,* where e is the number of edges in the flow graph using modifications of standard algorithmic design techniques [1, 9].

The notion of an S-region is similar to that of a statement in a well-structured language without the minimality condition, however, in addition to dealing with arbitrary program constructs we also disallow certain well-structured statements. The S-regions S_0 and S_2, for example, are very unstructured. Note too in the example that the set of blocks $\{B_{12}, B_{13}, B_{14}, B_{15}\}$ is not an S-region, although under certain definitions it might be considered a statement. S-regions were introduced in graph theory [10], where they were called hammocks, but have remained unstudied with the exception of one other compiler application [11].



S_0 consists of $B_2 - B_5$ with entry B_2 and exit B_5
S_1 consists of $B_5 - B_7$ with entry B_5 and exit B_7
S_2 consists of $B_7 - B_{10}$ with entry B_7 and exit B_{10}
S_3 consists of $B_{11} - B_{20}$ with entry B_{11} and exit B_{20}
S_4 consists of $B_{16} - B_{19}$ with entry B_{16} and exit B_{19}
S_5 consists of $B_{20} - B_{22}$ with entry B_{20} and exit B_{22}
S_6 consists of $B_1 - B_{23}$ with entry B_1 and exit B_{23}

FIGURE 1 A flow graph.

3 THE USE OF S-REGIONS IN SHACOOF

The decomposition into S-regions provides a mechanism for eliminating remaining delays (no-ops) that result from branches and loops. For an *S-region* S having *entry* x and *exit* y, the candidate subsequence P of instructions to be suppressed comprises:

1. The last instruction b of the entry block x if b is a branch statement
2. The label l at the top of the exit block y if l exists**
3. All instructions in all remaining blocks of S

By suppressing P, regarding it as one pseudo instruction, we can then apply a basic block scheduler (like that in reference 6) to the straight line code consisting of the block x (without b if $b \in P$), the single pseudo instruction P, and the block y (without l if $l \in P$).

A plurality of candidate subsequences can be identified for suppression in this way, by generating all S-regions, or generating them on demand, and replacing successively larger portions of the flow graph by such pseudo instructions when delays still remain, resulting in new, larger sections of straight line code that can be scheduled by the basic block scheduler.

For the example in Figure 1, the chain [2, P_0, 5, P_1, 7, P_2, 10] (where P_0, P_1, and P_2 represent the pseudo instructions for the S-regions S_0, S_1, and S_2, respectively) is straight line code, and it now becomes possible to move instructions from 7 up to 2 or from 5 down to 10, etc., assuming, in the normal manner, that data dependency permits. Note that the instruction subsequence for S_5 (which the pseudo instruction P_5 will replace) consists of only instructions from basic block 21, because basic block 20 has no branch and basic block 22 has no label.

4 IMPLEMENTATION ISSUES

The SHACOOF instruction scheduler forms part of an experimental version of one of the IBM XL family of compilers for the IBM RISC System/6000 computers. It can be called in to operation one or more times as required during compilation and is applied to instructions at an IL level. It can be used either conservatively (only after register allocation) or aggressively* (also before register allocation). On the SPEC Benchmark program EQNTOTT, it provides a further 6% run-time speedup over basic block scheduling alone.

The control flow graph provides the information on the structure of the basic blocks that is needed for the identification of the S-regions including the determination of their respective entry node x, exit node y, and intermediate nodes. An S-region table can be built in which data on x, y and the intermediate nodes for each S-region are stored, and which *implicitly* tag the instructions of the S-region. In other words, by accessing the S-region table, the instructions that belong to a can-

** We assume an IL for which every basic block begins with a label (with no instruction) only if there is a need for it, such as being the target of a branch.

* As with many compiler optimization techniques involving code motion, SHACOOF instruction scheduling may tend to increase register pressure by lengthening the "live" area of some variables. Therefore, very aggressive instruction scheduling applied *before* global register allocation may cause extra spill code to be inserted and may limit certain other optimizations such as coalescing [12–15].

Table 1. An Example of a Program Explaining the Operation of SHACOOF

```
   sample()
   {
1.    int i, count;
2.    count = 0;
3.    for(i=0;i< 10 ;i++
4.    {
5.     if (function1(i)) {count += 1;}
6.    }
      return count;
   }
```

didate subsequence for suppression can be identified. As the instructions are not physically tagged, there are no tags to be removed after scheduling.

It is common practice in modern compilers to maintain the sequence of IL instructions as a type of linked list. Before calling the SHACOOF scheduler, however, we modify the pointers in the instruction sequence list so that the basic blocks occur in the logical sequence dictated by the control flow graph, so that the entry block of an S-region precedes all intermediate blocks that in turn precede the exit node. This neither modifies the content of the instruction fields (merely the pointers accompanying them) nor modifies the control flow graph. After this modification, it is a simple matter to sequence through all the instructions of an S-region from the entry node to the exit node.**

** It should be noted that the order of the instructions within a basic block in the input instruction sequence, determined by the pointers, will correspond to the logical sequence in accordance with control flow. This will not normally be the case over basic block boundaries, due to branch and jump instructions between basic blocks.

Table 1 illustrates a sample program for scheduling and Table 2 gives the intermediate level machine instruction sequence corresponding to line 2 until line 6 of the program, annotated with instruction numbers (I.1–I.12) and basic block numbers (B_1 – B_4). A three-cycle delay needs to be covered between each *compare* instruction C and its corresponding conditional branch BT. The conventional basic block scheduler has succeeded in block B_2 to push up the add immediate AI which increments the loop index, thus covering only one cycle of the (I.6, I.8) pair. Nothing could be done for the (I.11, I.12) pair by the basic block scheduler.

SHACOOF recognizes that the set of basic blocks $\{B_2, B_3, B_4\}$ form an S-region with entry at B_2 and exit at B_4. The three instructions labeled with a plus sign indicate the subsequence P which is to be suppressed, that is, the last instruction of basic block B_2 (which is a branch), all instructions of basic block B_3 (in this example there is only one), and the first instruction of basic block B_4 (which is a label). Table 3 illustrates the instruction sequence as perceived by the basic block scheduler after P has been suppressed, called here "PSEUDO" and numbered SUPR. Following the suppression of instructions I.8 to I.10, the new instruction sequence from I.4 to I.12 in Table 3 forms a piece of straight line code that basic block scheduler handles in a conventional manner, resulting in the code illustrated in Table 4.

Instruction I.11 has been moved across the PSEUDO instruction in order to cover the pipeline delay between the *compare* of I.11 and the *branch* of I.12. The final output instruction sequence with the suppressed instructions restored is shown in Table 5. Not only has the pair (I.11, I.12) been covered, but the pair (I.6, I.8) that was originally

Table 2. An Annotated Sequence of Machine Instructions Corresponding to Lines 2–6 of the Program of Table 1

					Basic Block
	I.1	LI	r31=0	Load Immed.	B_1
	I.2	LR	r30=r31	Load Reg.	B_1
	I.3	CL.0:		label	B_2
	I.4	LR	r3=r30	Load Reg.	B_2
	I.5	CALL	r3=function1,1,r3	Call	B_2
	I.6	C	cr0=r3,0	Compare	B_2
	I.7	AI	r30=r30,1	Add Immed.	B_2
+	I.8	BT	CL.3,cr0,0x4/eq	Branch True	B_2
+	I.9	AI	r31=r31,1	Add Immed.	B_3
+	I.10	CL.3:		label	B_4
	I.11	C	cr1=r30,10	Compare	B_4
	I.12	BT	CL.0,cr1,0x1/lt	Branch True	B_4

Table 3. The Sequence of Instructions Following Suppression

I.1	LI	r31=t
I.2	LR	r30=r31
I.3	CL.0:	
I.4	LR	r3=r30
I.5	CALL	r3=function1,1,r3
I.6	C	cr0=r3,0
I.7	AI	r30=r30,1
SUPR	PSEUDO	r31,cr0
I.11	C	cr1=r30,10
I.12	BT	CL.0,cr1,0x1/lt

Table 4. The Sequence of Instructions Following Scheduling

I.1	LI	r31=0
I.2	LR	r30=r31
I.3	CL.0:	
I.4	LR	r3=r30
I.5	CALL	r3=function1,1,r3
I.6	C	cr0=r3,0
I.7	AI	r30=r30,1
I.11	C	cr1=r30,10
SUPR	PSEUDO	r31,cr0
I.12	BT	CL.0,cr1,0x1/lt

Table 5. The Output Sequence of Instructions

I.1	LI	r31=0
I.2	LR	r30=r31
I.3	CL.0:	
I.4	LR	r3=r30
I.5	CALL	r3=function1,1,r3
I.6	C	cr0=r3,0
I.11	C	cr1=r30,10
I.7	AI	r30=r30,1
I.8	BT	CL.3,cr0,0x4/eq
I.9	AI	r31=r31,1
I.10	CL.3:	
I.12	BT	CL.0,cr1,0x1/lt

covered by only one cycle is now covered by two cycles.

5 SUMMARY

The reordering of selected instructions at compile time can exploit the potential parallelism inherent in the code. In order to take most advantage of pipelined architectural features, we have presented the SHACOOF technique, which enlarges the "vision" of an instruction scheduler beyond basic blocks. This technique augments an already good basic block scheduler and extends its capability in covering delays with only minimal changes to existing compilers.

REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools.* Reading, MA: Addison-Wesley, 1986.

[2] D. Bernstein, H. Boral, and R. Y. Pinter, "Optimal chaining in expression trees," *IEEE Trans. Comput.* vol. 37, pp. 1366–1374, 1988.

[3] P. B. Gibbons and S. S. Muchnick, "Efficient instruction scheduling for a pipelined architecture," *Proc. ACM Symp. Compiler Construction,* vol. 21, pp. 11–16, 1986.

[4] J. R. Goodman and W.-C. Hsu, *Proceedings of the International Conference on Supercomputing.* St. Malo, France: ACM Press, pp. 442–452, 1988.

[5] J. L. Hennessy and T. Gross, "Postpass code optimization of pipeline constraints," *ACM Trans. Program. Languages Systems,* vol. 5, pp. 422–448, 1983.

[6] H. Warren, "Instruction scheduling for the IBM RISC System/6000 Processor," *IBM J. Res. Dev.,* vol. 34, pp. 85–92, 1990.

[7] S. Weiss and J. E. Smith, *Proceedings of the Second International Conference on Architectural Support for Programming Languages Operating Systems.* Palo Alto, CA: IEEE Press, 1987, pp. 105–109.

[8] M. C. Golumbic and V. Rainish, "Instruction scheduling beyond basic blocks," *IBM J. Res. Dev.,* vol. 34, pp. 93–97, 1990.

[9] M. C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs.* New York: Academic Press, 1980.

[10] V. N. Kas'janov, "Distinguishing hammocks in a directed graph," *Sov. Math. Dok.,* vol. 16, pp. 448–450, 1975.

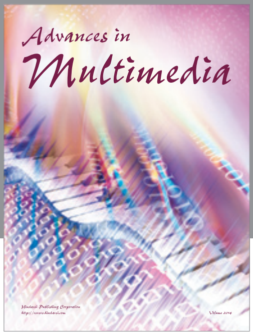
[11] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Sys.,* vol. 9, pp. 319–349, 1987.

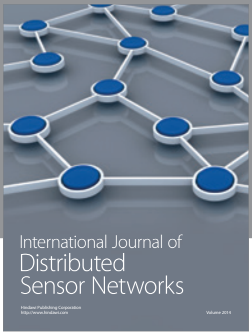
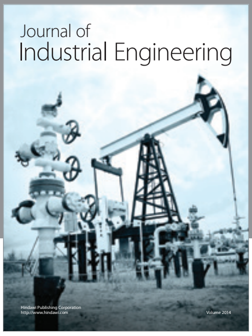
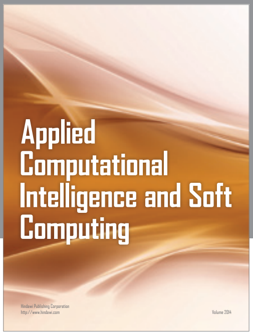
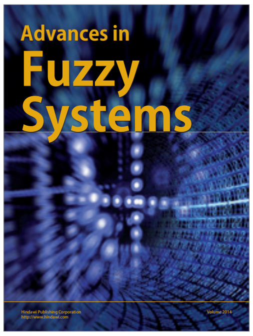
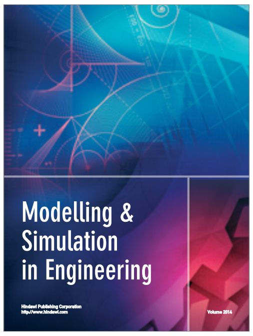
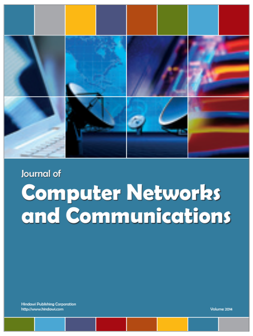
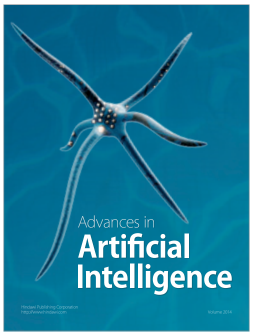
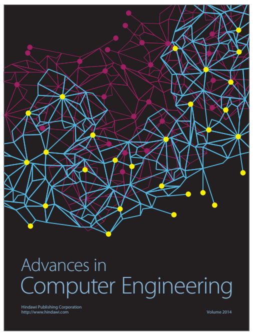
[12] M. A. Auslander and M. E. Hopkins, "An overview of the PL.8 compiler," *Proc. ACM Symp. Compiler Construction,* Vol. 17, pp. 22–31, 1982.

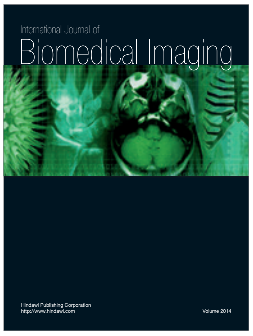
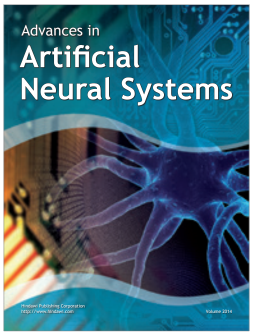
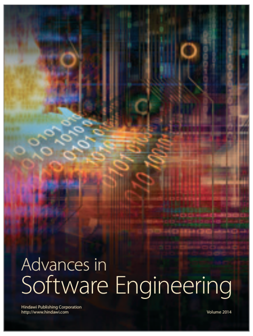
[13] D. Bernstein, D. Q. Goldin, M. C. Golumbic, H. Krawczyk, Y. Mansour, I. Nahshon, and R. Y. Pinter, "Spill code minimization techniques for optimizing compilers," *Proc. ACM SIGPLAN'89 Conf. Program. Language Design Implement.,* pp. 258–263, 1989.

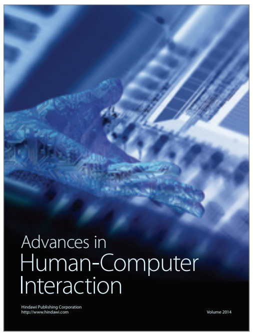
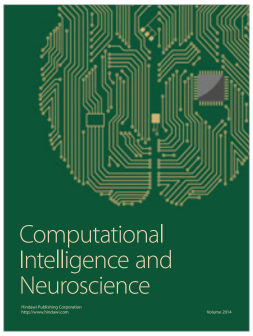
[14] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, Register allocation via coloring," *Comput. Lang.,* vol. 6, pp. 47–57, 1981.

[15] G. J. Chaitin, "Register allocation and spilling via graph coloring," *Proc. ACM Symp. Compiler Construction,* vol. 17, pp. 98–105, 1982.



Hindawi

Submit your manuscripts at
http://www.hindawi.com