Instruction scheduling for the IBM RISC System/6000 processor

by H. S. Warren, Jr.

For fast execution on the IBM RISC System/6000* processor, instructions should be arranged in an order that uses the arithmetic units as efficiently as possible. This paper describes the scheduling requirements of the machine, and a scheduling algorithm for it that is used in two compilers.

Introduction

The IBM RISC System/6000* processor is capable of executing as many as four instructions each cycle: two by the instruction-cache unit (ICU), one by the fixed-point unit (FXU), and one by the floating-point unit (FPU). If the "floating-point multiply-add" instruction is counted as two, it is capable of executing five instructions at a time. However, there are often delays, or hold-offs, between instructions. These arise primarily because of off-chip communication and the pipelined nature of the floating-point unit. For example, in executing a *load* instruction there is a hold-off of one cycle because the data must be fetched from the data cache, which is not on the FXU chip. Floating-point pipeline delay occurs if the result of one floating-point instruction is needed by the very next one. This delay occurs because the floating-point unit requires two cycles to complete the execution of a single instruction, but through pipelining it can execute a stream of instructions at a rate of one per cycle if they are independent (if the result of one is not needed by the very next instruction).

The purpose of the instruction scheduler is to arrange the instructions in an order in which they execute the fastest. It processes the output of the optimizer; code motion, commoning, strength reduction, etc., have been done. The output of the scheduler is passed to the final-assembly part of the compiler. We have found it desirable to run the scheduler twice: once before register allocation and once after. Instructions have much more freedom of movement before register allocation than after, and scheduling has its greatest effect during the first pass. It is run after register allocation because the register allocator may insert "spill" code (loads and stores), which should be scheduled, and because it makes other minor alterations of the code such as deleting "move register" instructions by its coalescing process.

The following sections discuss the scheduling requirements of the RISC System/6000 processor, and the scheduling algorithm that is currently in use in the XL compiler family [1] and in an earlier version of that compiler known as the PL.8 compiler [2]. The same scheduler is used for all XL languages.

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

Requirements

Fixed- and floating-point alternation The RISC System/6000 fixed- and floating-point units operate in parallel, and each is capable of executing one instruction per cycle. The branch unit is capable of sending out two instructions per cycle. Both instructions are sent out to both execution units, and each execution unit discards the instructions that are not appropriate to it. This means that for optimum system performance, the branch unit should send out one fixed-point and one floating-point instruction in each cycle. Actually, because of multicycle instructions such as floating-point divide (19 cycles) and fixed-point multiply (three to five cycles), the true optimal ordering of a long sequence of instructions is harder to describe. Because of buffering in the arithmetic units, however, it is almost always adequate to simply alternate them, which is what the XL compiler tries to do. If F denotes a floating-point instruction and X denotes a fixed-point instruction, an order such as FXXFFXX . . . would be just as good.

Load delay There is a delay of one cycle in the fixed-point unit between a fixed-point load and the first use of the data loaded. An independent fixed-point instruction (one that does not use the data loaded) should be placed between the two if possible. If no suitable instruction can be found, it is not necessary to insert a "no-op" because the hardware has interlocks on the registers and will wait when necessary.

For a *floating*-point load, there is no delay if the use is a floating-point store instruction. For other uses, there is usually no delay, because floating-point loads are processed by the fixed-point unit, and that unit usually runs ahead of the floating-point unit. However, in instruction traces one occasionally sees a delay here, and the present scheduler does try to insert an independent instruction.

A delay also occurs between a load and the first instruction to *set* the same register as the target of the load, but that is an unusual code sequence because the item loaded is discarded.

Because of the desirability of alternating the fixed- and floating-point instructions, there should also be two floating-point instructions between a fixed-point load and the first use. That is, if L denotes the load, X denotes an independent fixed-point instruction, Y denotes the fixed-point instruction that L feeds, and F denotes a floating-point instruction, then the optimal order is LFXFY. The same consideration applies to floating-point delays, *mutatis mutandis.*

Floating-point def-use delay There is a delay of one or two cycles in the floating-point unit between any floating-point instruction and the first use of the computed quantity. The delay is two if the first instruction feeds the FRA position (bits 11–15) of the second, and is otherwise one. One or two independent floating-point instructions should be inserted if possible. The extra delay for feeding the FRA position is due to the lack of a bypass path to the FRA position [3].

Several of the floating-point instructions have another field that can be interchanged with FRA without changing what the instruction does (the other field is FRB for *floating add*, and FRC for *floating multiply* and the *floating-multiply-add* family of instructions). Thus, the extra delay can often be avoided by interchanging the appropriate fields.

Compare-branch delay A complete description of the delays associated with conditional branches would be quite complicated, so we describe only the main effects.

Between a fixed-point *compare* and a conditional branch (*bc*) on the condition register (CR) field that the *compare* set, there is a delay of three cycles if the branch is taken, and usually no delay if the branch falls through. In the case of a floating-point *compare*, the delay is six cycles if the branch is taken, and three if it falls through. These delays occur in both the fixed-point and floating-point units.

To understand the timing of conditional branches, it is helpful to understand a little about how the branch unit operates. As the branch unit sends out instructions, it decodes them sufficiently to tell whether they set or use the CR, and if so, which CR field is affected. When it sends out a *compare*, it locks the target CR field. When it sends out an instruction with the "record" bit set, it locks either CR0 or CR1, depending upon whether the instruction is fixed-point or floating-point. Later, when the instruction is executed by its execution unit, the branch unit receives the CR value. It then unlocks the CR field.

When the branch unit executes a conditional branch instruction, it waits, if necessary, for the tested CR field to be unlocked. However, if the CR field is locked, it "conditionally dispatches" two instructions. This is sufficient to keep the fixed-point unit busy if the branch falls through, and if the two conditionally dispatched instructions and the two instructions following those are all fixed-point instructions.

If, when a CR field is locked, another instruction is encountered that sets that CR field, then the branch unit stops sending out instructions, because it cannot handle two or more outstanding instructions that set the same CR field. The programmer should try to avoid such sequences. For example, consider the following:

```
andil. . . .      (Sets CR0 implicitly)
cmp  cr0, . . . (Also sets CR0)
bc   cr0, . . . (Branch uses CR0)
```

Both the *andil.* and the *cmp* set CR0, so there will be a delay of three cycles between them (there is an additional three-cycle delay between the *cmp* and the *bc* if the branch is taken). The programmer can avoid the unnecessary delay by using a different CR field for the *cmp-bc*. As another example, in the sequence

```
cmp cr0, . . . (Sets CR0)
bc   cr0, . . . (Branch uses CR0)
cmp cr0, . . . (Sets CR0)
bc   cr0, . . . (Branch uses CR0)
```

the branch unit will not conditionally dispatch the second *cmp*. This makes the sequence run unnecessarily slowly in the case that the first *bc* falls through. The programmer should use different CR fields in the first and second *cmp-bc* sequences.

Because of the CR locking, the processor can be slowed substantially by indiscriminate use of the "record" bit. The record bit should be set only when the CR0/CR1 result is actually needed.

The CR fields are "paired" for the purposes of locking. The pairs are CR0 and CR4, CR1 and CR5, CR2 and CR6, and CR3 and CR7. When an instruction sets CR0, for example, the branch unit actually locks both CR0 and CR4. Thus, in the first code sequence shown above, the programmer should use some CR field other than 0 or 4 for the *cmp-bc*. Similarly, in the second code sequence, the CR fields should be chosen from different pairs. This "pairing" is necessitated by timing problems and the limited fan-in of the CMOS-2 technology, and does not affect the logical operation of a program.

The machine will be slowed if it has to execute branches rapidly in succession. For example, for a *bc* that falls through to another *bc*, the machine waits for the first *bc* to be resolved before it executes the second. That is, conditional dispatching has been blocked. This occurs even if the second branch is unconditional. For an unconditional branch to a conditional branch that falls through, however, there is usually no delay. The programmer can take advantage of this by rearranging loops that end in *cmp-bc* as follows (see also [4]):

```
                                          b     in
                                   loop:  bc¬   crn, out
loop:  . . .                       in:    . . .
       . . .              ⇒              . . .
       cmp  crn, . . .                   cmp   crn, . . .
       bc   crn, loop                    b     loop
                                   out:
```

The first loop will have a three-cycle delay, and the second will have no delay, provided the code shown as ". . ." amounts to at least four fixed-point cycles. By "bc¬ " above we mean the original *bc* instruction with the sense reversed. Of course, if two or three instructions can be found to place between the *cmp* and the *bc*, or if the loop can be changed to use a branch on the count register rather than on the CR, that is preferable.

Excessive floating-point stores (stfs) A floating-point store instruction normally takes one cycle in both the fixed- and floating-point units. However, if there are more than four consecutive *stfs*, there *may* be a delay of one cycle, in the fixed-point unit, for each *stf* after the fourth.

This delay is caused by the fact that the fixed-point unit may compute addresses faster than the floating-point unit can compute the quantities to be stored. There is a "pending store queue" of length four that holds the virtual addresses of floating-point data to be stored. If the queue is full, further *stfs* are held up until a slot in the queue becomes available.

Store-load-use delay The load-use delay is magnified by the presence of a store preceding the load, with the effective addresses referring to the same 16-byte block of memory. The length of the delay depends upon whether the instructions are fixed- or floating-point, as shown below. ST and L denote fixed-point store and load instructions, and STF and LF denote floating-point store and load instructions.

ST- L-use: 3 ST- LF-use: 2

STF-L-use: 5 STF-LF-use: 5

The delay is in the fixed-point unit for L and in the floating-point unit for LF. It occurs between the load and the use of the data loaded, but it can be ameliorated by including instructions either between the store and the load or between the load and the use.

The STF-L and ST-LF sequences are of some importance because they are used for conversions between fixed- and floating-point instructions.

mfspr and mfcr delay There is a delay of one cycle in the fixed-point unit between *move from special-purpose register* or *move from condition register* and the first use of the data moved to a GPR. The delay length is the same no matter what the special-purpose register is (even if it is on the fixed-point chip).

mtlr-br delay The sequence *move to link register* followed by *branch (un)conditional register* is often used for subroutine return and for branches to targets that are variable. The delay here is four cycles.

Leaf subroutines can normally avoid this delay by leaving the return address in the link register, so the *mtlr* is not required. For these, the *br* instruction normally takes no time at all, as it is executed in parallel with arithmetic unit processing. More complex subroutines often have a *load multiple* instruction in their epilog. This can be placed between the *mtlr* and the *br*; if the *load multiple* loads four or more registers, the *br* is again free. Hence it is only those subroutines of intermediate complexity that suffer this delay.

mtctr-bct delay Between *move to count register* and a *bc* that uses the count register, there is a four-cycle delay.

CR-logic instruction delays The RISC System/6000 instruction set includes a complete set of logic instructions for manipulating bits of the CR, e.g., *crand*, which forms the logical *and* of the bits at two CR positions and puts the result in a third bit position. These instructions are executed by the branch unit. The presence of several consecutive instructions of this type may cause delays in the execution units, because the branch unit will not be able to deliver instructions fast enough. In a sequence of alternating fixed- and floating-point instructions, having more than two consecutive CR-logic instructions will cause a delay in each arithmetic unit. In a sequence of fixed-point instructions, having more than five consecutive CR-logic instructions will cause a delay in the fixed-point unit.

There is no delay for a CR-logic instruction followed immediately by a conditional branch on the bit set.

Moving to and from the FPSCR Executing *move to floating-point status and control register fields* (*mtfsf*) causes a pipeline drain in the floating-point unit. This causes a delay of two cycles. However, the delay cannot be covered by floating-point instructions, so it really acts as if *mtfsf* were a three-cycle instruction.

The immediate forms (*mtfsi*, *mtfsb1*, and *mtfsb0*), however, do not cause a pipeline drain nor any delay; they are simply one-cycle instructions.

There is a normal floating-point delay between *move from FPSCR* (*mffs*) and the first use of the target register (two cycles if it feeds the FRA position, one cycle otherwise).

Other scheduling considerations

Minimizing "liveness" Scheduling tends to increase register pressure, so it is important not to "overschedule." That is, loads should not generally be moved back farther than necessary. As a simple example of this, suppose a basic block consists of three load-store pairs. We consider three possible orders of this below, where the code is shown before register allocation (i.e., "r101" denotes a symbolic register that has not yet been assigned to a physical register):

```
L   r101,...    L   r101,...    L   r101,...
ST  r101,...    L   r102,...    L   r102,...
L   r102,...    L   r103,...    ST  r101,...
ST  r102,...    ST  r101,...    L   r103,...
L   r103,...    ST  r102,...    ST  r102,...
ST  r103,...    ST  r103,...    ST  r103,...
      (a)             (b)             (c)
```

Order (a) is how the code would probably be arranged before scheduling, assuming it was produced by three consecutive assignment statements in the source program. This order takes nine cycles to execute, because of the one-cycle delays between each *load* and *store*. Order (b) executes in only six cycles, but it suffers from "overscheduling" in that the register allocator will have to assign three distinct physical registers to r101, r102, and r103. Order (c) also executes in only six cycles, but requires only two physical registers, as r101 and r103 can be assigned to the same register. Thus (c) is the best order.

On the other hand, the possibility of cache misses suggests that loads should be moved as far back as possible. This is a difficult trade-off to make.

Incidentally, this example also shows the value of scheduling before registers are allocated. If it were not done then, sequence (a) would be the input to the register allocator, and it would most likely assign r101, r102, and r103 to the same physical register, preventing any rearrangements.

Avoiding semantic changes Floating-point instructions should generally not be moved across subroutine calls, because the floating-point instructions set status bits in the FPSCR, and the called subroutine may be testing them.

Many instructions cannot be safely moved back across conditional branches, even though on the surface it may seem valid, because the conditional branch may be preventing the execution of the instruction when conditions are such that executing it would cause an error condition. This applies to most loads and stores (which can cause various forms of the *data storage interrupt*), and to the floating-point instructions (because of the FPSCR). Fixed-point instructions other than loads and stores may be freely moved provided the OE bit is off (if the OE bit is on, movement may be undesirable because of the *summary overflow* bit in the XER).

The leveling algorithm

The instruction-scheduling problem has its origin in microprogramming. Microcoded machines can often

execute two or more instructions simultaneously by packing them into the same microinstruction, provided there are no data dependencies between them, and they are appropriate types of operations. The "microcode compaction" problem, as it is called, is very similar to the problem of arranging the instructions in an optimal order in the presence of hold-offs. Reference [5] is a good review of the state of the art about ten years ago.

The scheduling algorithm used in the XL and PL.8 compilers is based on a "dependency graph," which is constructed for each basic block. The dependency graph has a node for each instruction, and a directed arc between two nodes if one of the instructions must precede the other for any reason. If the first instruction computes something used by the second, that is called a "forward dependency." If the second instruction alters something (a register or storage) used by the first, that is called a "reverse dependency." Most delays occur for forward dependencies. The arcs in the dependency graph are labeled with the amount of the delay, or hold-off, between the instructions.

As an example, consider the basic block containing the following source code (fixed-point arithmetic):

A = B + C − D;

IF E > 0 THEN . . .

for which the intermediate language code (before register allocation) is the following:

```
L     r100, B(r200)
L     r101, C(r200)
ADD   r102, r100, r101
L     r103, D(r200)
SUB   r104, r102, r103
ST    r104, A(r200)
L     r105, E(r200)
CMP   r106, r105, 0
BC    r106, . . .
```

This has the dependency graph shown in **Figure 1**, where the notations *L.B*, *L.C*, etc., denote a load of *B*, *C*, etc. Although the ST instruction has no forward nor reverse dependency with the *bc* (conditional branch), an arc is placed between them to reflect the fact that the ST (and in fact all the instructions in the basic block) must be executed before the *bc*.

The scheduler constructs the dependency graph by examining all the $(n^2 - n)/2$ pairs in the n-instruction basic block. For each instruction pair, the registers are examined to see whether or not one instruction sets a register that the other one sets or uses. Although a simple hashing technique is used to speed up the register matches, construction of the dependency graph remains the most time-consuming part of the scheduler.

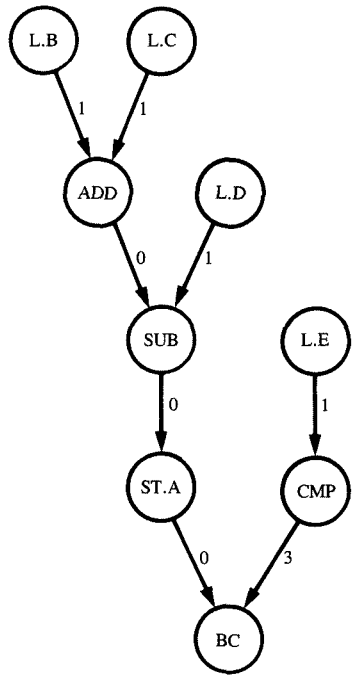

Figure 1

Example dependency graph.

The pair of instructions are also examined to see if they interfere through storage. The storage classes and other dictionary information are used to see whether the instructions cannot possibly refer to the same storage location (e.g., the STATIC and AUTOMATIC classes cannot overlap). If they are in the same or possibly overlapping classes, the base register and displacement are examined to see if they definitely refer to different storage locations. This handles cases such as a store into A(I) followed by a load from either A(I + 1) or A(I − 1); in this case there is no storage dependency, and it might very well be advantageous to place the load before the store.

After the dependency graph has been constructed, it is walked upward and each node is labeled with the maximum delay from the node to the end of the basic block. This is calculated by adding the delay times from the end of the basic block up to each node. If two or more paths converge on the same node (a situation not shown in Figure 1), the maximum value calculated along the paths is used.

Next, instructions are selected in, basically, decreasing order of the delay times from each node to the end.

Referring to Figure 1, the four "load" instructions are eligible in that they have no predecessors in the dependency graph. The one with the largest delay to the end of the basic block is chosen first. This is the load of *E*, which has a four-cycle delay to the end of its path, whereas the other loads have only a one-cycle delay to the end of their paths.

As instructions are selected, the algorithm updates a "current time" value, which is initially zero, by incrementing it by the execution time of the instruction just selected (the execution time is one cycle for most instructions). The instruction selected is removed from the dependency graph.

Also, as each instruction is selected, its immediate successors in the dependency graph are marked with a time value equal to the updated current time plus the delay from the instruction just selected to the successor. This "earliest time" value is used to hold off instructions, if possible, until after the required delay has elapsed.

Referring again to Figure 1, after putting out the load of *E* at time 0, the CMP is marked with an earliest time of 2. Now the algorithm looks for an instruction that has no successors in the dependency graph, and whose earliest time is less than or equal to the current time (1). The loads of *B*, *C*, and *D* satisfy this. One is chosen nearly arbitrarily; the load of *B* is chosen because it occurred first in the original code. The current time is updated to 2, and the ADD is marked with an earliest time of 3.

Next, the instructions that have no predecessors are the two remaining loads, which have an earliest time of 0, and the CMP, which has an earliest time of 2. The CMP can be selected, because its earliest time has been reached. Of the three instructions that are equally good candidates at this point, the CMP is chosen because it has the greatest delay to the end of its path (three cycles).

Next, only the loads of *C* and *D* are eligible in the sense of having no predecessors. The algorithm makes the nearly arbitrary choice of *C*. The "earliest time" for the ADD is changed to 5, since the load of *C* was selected at time 3.

The algorithm continues in this way, and the final order of the instructions is

```
L     r105, E(r200)
L     r100, B(r200)
CMP   r106, r105, 0
L     r101, C(r200)
L     r103, D(r200)
ADD   r102, r100, r101
SUB   r104, r102, r103
ST    r104, A(r200)
BC    r106, . . .
```

The algorithm found an optimal order. There are no uncovered delays, and the above code executes in eight cycles on the RISC System/6000 processor.

However, the strategy of choosing first the instruction that is farthest from the end in terms of delay along its path is merely a heuristic and does not always get the best order. As an example of where it fails, consider Figure 1 with the loads of *D* and *E* omitted. The algorithm would then choose the CMP first. Then it would be forced to choose the two loads and the ADD, and there would be an uncovered delay between the second load and the ADD. However, the best order would place the two loads first, then choose the CMP as a cover for the second load, and then choose the ADD, SUB, ST, and BC. There are just enough instructions to cover the CMP-BC delay of 3, so there is no delay with this ordering.

Refinements to the basic algorithm

The complete algorithm employs a few other criteria to determine the ordering. There is no "planning" or "lookahead" associated with these other criteria; they are merely used as tie-breakers when choosing the instruction to select next. The complete selection scheme is described below. For ease of exposition it is described as a subsetting process, although it is implemented by scanning the remaining instructions, choosing the "best" one to select. The scan is done once for each instruction selected.

1. Initialize the set of all those instructions that have not yet been selected, and that have no predecessors in the dependency graph (these are the "legal" ones).
2. Refine the subset to those instructions whose earliest time has arrived or, if none, those with the smallest earliest time.
3. If one or more instructions have been selected, and if the current subset contains one or more instructions of opposite type (fixed/floating) from the last one selected, then refine the current subset to those of this opposite type.
4. Refine the subset to those of maximum total delay along the path of each instruction to the end of the basic block.
5. Refine the subset to those of minimum "liveness weight."
6. Refine the subset to those with greatest "uncovering."
7. Refine the subset to the unique instruction that came first in the original ordering.
8. Select the single instruction that is in the subset at this point, and then repeat this process until all instructions in the basic block have been selected.

The "liveness weight" criterion is used for a simple scheme that reduces register spills. Each instruction is

assigned a "weight" as follows:

0 Move-register (fixed-point or floating-point)
1 Instructions with no target registers (stores and traps)
2 Most instructions
3 Loads (from storage)
4 Instructions with no source registers (e.g., Load Immediate)

The "lightweight" instructions are selected first (if there is a tie for all the more important criteria). The *move register* instructions have the lightest weight for a technical reason: The register allocator does a better job of coalescing if these are kept close to the instruction that defines the source of the *move register*. Stores and traps are selected early because they have "uses" but no "definitions." Hence, they may free up registers, but they never increase the number that are currently live. Similarly, an instruction such as *load immediate* is selected late, because it increases the number of registers that are live and does not free any.

The weights are stratified with respect to subroutine CALLs. That is, instructions between the beginning of the basic block and the first CALL are given weights from 0 to 4; those between this CALL and the next have their weights increased by ten, etc. This has the effect of making it less likely that instructions will be moved across CALLs. Our experience has been that moving instructions across CALLs tends to increase the number of times a definition and its use bridge a CALL; such a bridge is undesirable because the CALL instruction has a large register "kill."

Reducing spills seems to be a very difficult problem. We tried some more complicated schemes, but the simple one described above seemed to work about as well as any.

The "uncovering" step mentioned above is an attempt to get better scheduling by having a wider choice at each step. If there is no choice between two instructions up to the point that this is considered, the algorithm computes how many instructions will become ready if either is selected next. By "ready" we mean here that the instruction has no predecessors in the dependency graph and its earliest time will have been reached. If there is a difference, the one is chosen that maximizes the number of instructions made ready. This heuristic tends to improve the schedule in a number of respects, particularly in the alternation of fixed- and floating-point instructions.

Other machines

We have described the instruction scheduler as it pertains to the RISC System/6000 processor. The same program also schedules for the 801 and ROMP-C [6]. To tailor it to these other machines is mainly a matter of adjusting a few parameters such as the delays and execution times. However, for the ROMP the scheduler also avoids having more than two consecutive load or store instructions, because of the presence of the "load-store queue," which is of length 2. For the ROMP and the 801, the scheduler also attempts to place an optimal instruction next to a branch, for "branch with execute" (delayed branch) generation.

Future directions

The current trend is toward computers with more scheduling requirements, i.e., more situations that require scheduling for good performance, and larger hold-offs. The high-quality compiler of the future will employ more sophisticated scheduling techniques than have been described here. Significant work has already been done in this area, particularly for the VLIW (very long instruction word) architectures [7, 8]. Theoretical work is presented in [9] and [10].

The existing XL compiler family will probably be extended so that the scheduler will move code across basic blocks when that gives an improvement. One approach toward this is discussed in this issue [4]. Other approaches are discussed in [11] and [12].

Acknowledgments

The original scheduler was designed by Richard Goldberg for the PL.8 compiler. That program handled the 801 and the original ROMP. The more complex scheduling requirements of the RISC System/6000 processor and ROMP-C necessitated a redesign, which was done by this author with the assistance of Richard Goldberg, David Bernstein, and Peter Hsu. Joanne B. Minish recoded it for incorporation into the XL compiler family.

References

1. AIX/RT FORTRAN Reference Manual, Order No. SC09-1267, available through IBM branch offices.
2. 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; also, M. E. Hopkins, "Compiling for the RT PC ROMP," *Reduced Instruction Set Computers,* Second Edition, IEEE Computer Society Press, Los Alamitos, CA, 1990, pp. 146–153.
3. R. K. Montoye, E. Hokenek, and S. L. Runyon, "Design of the IBM RISC System/6000 Floating-Point Execution Unit," *IBM J. Res. Develop.* **34,** 59–70 (1990, this issue); G. F. Grohoski, "Machine Organization of the IBM RISC System/6000 Processor," *IBM J. Res. Develop.* **34,** 37–58 (1990, this issue).
4. M. C. Golumbic and V. Rainish, "Instruction Scheduling Beyond Basic Blocks," *IBM J. Res. Develop.* **34,** 93–97 (1990, this issue).
5. David Landskov, Scott Davidson, Bruce Shriver, and Patrick W. Mallett, "Local Microcode Compaction Techniques," *Comput. Surv.* **12,** 261–294 (September 1980).
6. John Cocke and V. Markstein, "The Evolution of RISC Technology at IBM," *IBM J. Res. Develop.* **34,** 4–11 (1990, this issue).

7. Joseph A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Trans. Computers* **C-30,** 478–490 (July 1981).
8. Kemal Ebcioglu, "Some Design Ideas for a VLIW Architecture for Sequential-Natured Software," *Research Report RC-13795*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, November 1987.
9. David Bernstein and Izidor Gertner, "Scheduling Expressions on a Pipelined Processor with a Maximal Delay of One Cycle," *ACM Trans. Prog. Lang. Syst.* **11,** 57–66 (January 1989).
10. D. Bernstein, J. M. Jaffe, R. Y. Pinter, and M. Rodeh, "Optimal Scheduling of Arithmetic Operations in Parallel with Memory Access," *Technical Report 88.136*, Israel Scientific Center, Technion City, Haifa, Israel, May 1985.
11. Alexandru Nicolau, "Percolation Scheduling: A Parallel Compilation Technique," *Technical Report 85-678*, Department of Computer Science, Cornell University, Ithaca, NY, May 1985.
12. Kemal Ebcioglu, "A Compilation Technique for Software Pipelining of Loops with Conditional Jumps (Preliminary Version)," *Proceedings of the 20th Annual Workshop on Microprogramming* (*Micro 20*), Colorado Springs, CO, December 1–4, 1987 (ACM Order No. 520870), pp. 69–79.

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

Henry S. Warren, Jr. *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.* Dr. Warren graduated from Rensselaer Polytechnic Institute in 1960 with the B.S.E.E. degree. He received his M.S. in mathematics and Ph.D. in computer science from New York University in 1964 and 1980, respectively. Dr. Warren joined IBM's Federal Systems Division in 1961, helping to develop several military command and control systems. He joined the Research Division in 1973, and has been a member of the 801 development group since 1978. Most of his recent work has been on the PL.8 compiler, for which work he received an Outstanding Contribution Award in 1981. Dr. Warren is a member of the IEEE Computer Society, Eta Kappa Nu, and Tau Beta Pi.