International Journal of Parallel Programming, Vol. 24, No. 3, 1996

Using Predicated Execution to Improve the Performance of a Dynamically Scheduled Machine with Speculative Execution

Po-Yung Chang,[1] Eric Hao,[1] Yale N. Patt,[1] and Pohua P. Chang[2]

Conditional branches incur a severe performance penalty in wide-issue, deeply pipelined processors. Speculative execution[1, 2] and predicated execution[3-9] are two mechanisms that have been proposed for reducing this penalty. Speculative execution can completely eliminate the penalty associated with a particular branch, but requires accurate branch prediction to be effective. Predicated execution does not require accurate branch prediction to eliminate the branch penalty, but is not applicable to all branches and can increase the latencies within the program. This paper examines the performance benefit of using both mechanisms to reduce the branch execution penalty. Predicated execution is used to handle the hard-to-predict branches and speculative execution is used to handle the remaining branches. The hard-to-predict branches within the program are determined by profiling. We show that this approach can significantly reduce the branch execution penalty suffered by wide-issue processors.

KEY WORDS: Predicated execution; speculative execution; branch prediction; wide-issue dynamically scheduled processors; instruction level parallelism.

1. INTRODUCTION

Today's processors are being built with wider issue rates and deeper pipelines in order to exploit larger amounts of instruction-level parallelism.

[1] The University of Michigan, Ann Arbor, Michigan.
[2] Intel Architecture Laboratory, Santa Clara, California.

0885-7458/96/0600-0209$09.50/0 © 1996 Plenum Publishing Corporation

For such processors, the occurrence of branches in the instruction stream incurs a severe performance penalty.(10, 11) Two well-known mechanisms have been proposed to reduce this penalty, speculative execution in conjunction with branch prediction and predicated execution.

Speculative execution(1, 2) is a microarchitectural mechanism that solves the branch problem by guessing the direction that a branch will take before the branch condition is known. After making a prediction for a branch in the dynamic instruction stream, the processor speculatively executes the instructions along the predicted path. The branch prediction is confirmed when the branch instruction is executed. If the prediction is correct, then the processor suffers no performance penalty for this dynamic instance of the branch. If the prediction is incorrect, the processor, after removing from its state the effects of the speculatively executed instructions, must return to the point of the branch prediction and begin executing instructions from the correct path. In this case, the full branch execution penalty is suffered.

Predicated execution(3–9) is an architectural mechanism that addresses the branch problem by providing the compiler with a set of predicated instructions that can be used to eliminate static branches in the program. The branches can be eliminated by replacing their control-dependent instructions with predicated instructions.(12) A predicated instruction contains an extra source operand known as the predicate operand. The predicated instruction is conditionally executed based on the value of this operand. If the predicate evaluates to *true*, the predicated instruction is executed like a normal instruction. If the predicate evaluates to *false*, the instruction is not executed. Given these semantics, a compiler can replace the branch and the set of instructions that represent an *if-then-else* statement by predicating the *then* clause with the branch condition as the predicate and predicating the *else* clause with the complement of the branch condition as the predicate. By eliminating the branch, predicated execution ensures that the processor never suffers any branch execution penalties due to that branch.

While both speculative and predicated execution have been shown to significantly reduce the performance penalty due to branches, both mechanisms have disadvantages. A processor using speculative execution suffers the full branch execution penalty whenever it makes a branch misprediction. A processor using predicated execution may see a drop in performance because predicated execution changes control dependencies into flow dependencies, lengthening the dependency chains in the program. In addition, using predicated execution wastes issue bandwidth because the predicated instructions from both branch paths must always be issued. Finally, predicated execution has the disadvantage that certain branches have characteristics that prevent them from being eliminated. A mechanism other than predicated execution must be used to handle such branches. [Two other disadvantages to using predicated execution are that it requires the addition of a third operand to every instruction in the instruction set architecture and the addition of hardware to support its semantics. These issues are not addressed in this paper.]

This paper examines the performance benefit of using both speculative and predicated execution to reduce the branch execution penalty for wide-issue, dynamically scheduled machines that use the Two-Level Adaptive Branch Predictor.(13–15) To minimize the effect of each approach's disadvantages, speculative execution is used to handle the branches that are accurately predicted by the branch predictor and predicated execution is used to eliminate the remaining hard-to-predict branches. Profiling is used to determine which branches are inaccurately predicted by the branch predictor. In addition, we examine the effectiveness of the *instruction promotion* compiler optimization,(6, 16) which removes the data dependencies that were created by the addition of predicated instructions. We show that for most of the SPECint92 benchmarks, profiling is an effective means for determining which branches are difficult to predict for the Two-Level Predictor. Furthermore, we show that the addition of predicated execution can provide a significant performance increase while the instruction promotion optimization can provide a small additional performance increase.

This paper is organized into five sections. Section 2 discusses previous work done with predicated execution. Section 3 discusses the methodology used to eliminate hard-to-predict branches. The predicated execution model used in this study and the results from profiling the benchmarks for hard-to-predict branches are presented. Section 4 describes the simulation methodology used in our experiments and the results from those experiments. Section 5 provides some concluding remarks.

2. PREVIOUS WORK

Many researchers have studied the effectiveness of combining speculative and predicated execution. Pnevmatikatos and Sohi(7) studied the performance benefit of using predicated execution in conjunction with the Two-Level Adaptive Branch Predictor. They eliminated as many branches as possible through predicated execution and reported its effect on the number of dynamic branches executed, basic block size, wasted issue bandwidth, and the dynamic window size (the number of instructions issued between branch mispredictions). Tyson(8) studied the performance benefit of predicating all short forward branches. In addition to the metrics reported in Ref. 7, he approximated the reduction in the branch execution penalty due to predicated execution by calculating the penalty as a function of processor issue width and pipeline depth. Mahlke *et al.*(6, 9) studied the performance benefit of using predicated execution in conjunction with a static branch predictor or a simple dynamic branch predictor.

In examining the performance benefit of using both speculative and predicated execution, this paper builds on the work of these previous researchers and makes three contributions. First, it examines the performance benefit of eliminating only the hard-to-predict branches for the Two-Level Branch Predictor. Because the Two-Level Branch Predictor achieves a much higher prediction accuracy than those examined in Refs. 6 and 9, the number of branches that must be eliminated is smaller. Second, it measures this performance benefit by simulating an actual dynamically scheduled machine and reporting the benefit both in terms of number of mispredictions eliminated and the number of cycles saved in total execution time. Third, it shows the effectiveness of using profiling to detect branches that are hard-to-predict for the Two-Level Branch Predictor.

3. PREDICATED EXECUTION

3.1. Predication Model

Our predication model assumes that each predicated instruction has a register destination and three or four source operands: one or two source operands for calculating the result generated by the instruction, one predicate operand, and one implicit source operand specified by the destination register.(7, 17) We have assumed that the predicate operand is an ordinary register, although we expect some future implementations to use Boolean registers for that purpose. The predicated instruction uses the least significant bit of a register as the value of its predicate.

Although predicate registers are usually set by compare instructions, they can be set by any instruction. If the predicate evaluates to *true*, the predicated instruction executes like a regular instruction: the destination register receives the result of the instruction's operation. If the predicate evaluates to *false*, the predicated instruction writes the old value of the destination register (the implicit operand) back into the destination register. This is done instead of suppressing the execution of the instruction because the machine simulated uses dynamic register renaming.(18, 19) For register renaming to operate correctly, every issued instruction having a destination register must produce a result. This requirement has the drawback that it forces every predicted instruction to be part of a dependency chain regardless of the value of its predicate.

In our predicated execution model, each predicated instruction is allowed to execute once the predicate value and the required operands are ready. That is, if the predicate is *true*, the predicated instruction needs to wait only for the value of the source operands. On the other hand, if the predicate is *false*, the predicated instruction needs to wait only for the old value of its destination register. With this execution model, the implicit data dependence usually causes only one additional cycle of delay.

Figure 1 illustrates this drawback with a modified code fragment from the eqntott benchmark. Although there is no true dependence between the two predicated move instructions in Fig. 1, the second move instruction has a data dependence on the first move instruction due to the implicit destination operand (r5). [Sprangle and Patt(20) have proposed a static register tagging scheme that avoids this drawback by eliminating the need to execute predicated instructions when their predicates are false. Our simulations, however, do not take advantage of this scheme.] When the second instruction is issued, the implicit destination register gets the old tag for r5, say x, and a new tag, say y, is assigned to the destination register r5. When the third instruction is issued, the destination register is assigned with a new tag, say z. Thus, if r4 is *true*, there will usually be one additional cycle of delay for instruction (2′) to pass the value of r5 to instruction (3′); instruction (3′) then can distribute the new value of r5 with tag z.

We should point out that there could be additional delay cycles if instruction (3′) is not able to execute in the next cycle, because other ready instructions are fired first. These additional delay cycles can usually be eliminated with the correct assignment of operations to node tables.

```
/* C code */
if (aa < bb)
   res = -1;
else
   res = 1;


;; Assembly code with predicated instructions
(1)   cmp_lt  r4, r2, r3
(2)   movi    r5, -1,    if r4
(3)   movi    r5,  1,    ifnot r4


;; Decoded instructions after register renaming
(1')  cmp_lt  r4 = tag w,  r2 = tag u,  r3 = tag v
(2')  movi    r5 = tag y,          -1,  r5 = tag x,  if r4 = tag w
(3')  movi    r5 = tag z,           1,  r5 = tag y,  if not r4 = tag w
```

Fig. 1. Elimination of an *if-then-else* branch with predicated instructions.

On the other hand, if r4 is *false*, instruction (3′) does not need to wait for tag y, resulting in no additional delays.

3.2. Branch Profiling

For static branch predictors, branch taken-rate can effectively identify the hard-to-predict branches. However, for dynamic branch predictors, branch taken-rate alone may not be sufficient in identifying the hard-to-predict branches. That is, dynamic predictors may be able to accurately predict those branches whose taken-rates are not mostly-taken or mostly-not-taken. Using taken-rate alone, we may mistakenly classify these branches as hard-to-predict. Figure 2 shows a branch with a dynamic taken-rate of 60%. This branch is taken for the first 6 times and then not-taken for the next 4 times. Using taken-rate as the metric, this branch will be classified as hard-to-predict. However, as shown in the lower part of the table, a simple dynamic predictor like the last time taken predictor can capture the change in this branch's behavior and predict this branch with 90% accuracy. Therefore, profiling the performance of the dynamic branch predictor may better identify the hard-to-predict branches.

To determine which branches to consider for elimination, each benchmark was profiled with a training data set. The profiler modeled the processor's branch predictor and recorded the number of mispredictions for each static branch. Branches whose misprediction counts exceeded a given threshold were considered hard-to-predict and marked for elimination.

The branch predictor simulated by the profiler in our experiments was the Gshare/PAg predictor,[15, 21] an aggressive variation of the Two-Level Branch Predictor. This hybrid branch predictor combines the global history (Gshare[21]) and per-address history (PAg[13]) schemes of the Two-Level Branch Predictor into one predictor. It bases its prediction on whichever scheme has recently been making the more accurate predictions for the branch to be predicted. An array of 1K 2-bit updown counters is

- Static predictors: record branch direction

Branch:	T	T	T	T	T	T	N	N	N	N
Profile:	T: 6, N: 4									

- Dynamic predictors: record correct branch predictions

Branch:	T	T	T	T	T	T	N	N	N	N
Predictions:	T	T	T	T	T	T	T	N	N	N
Profile:	C: 9, I: 1									

C = correct prediction, I = incorrect prediction

Fig. 2. Branch profiles based on taken-rate and prediction accuracy.

used to keep track of the relative accuracies of the two schemes. Both the Gshare and the PAg schemes use 10-bit branch history registers and 1K-entry pattern history tables for making their predictions.

The profiler simulates the behavior of the branch predictor by modifying the program to be profiled. It replaces the code that calculates the branch conditions for each branch in the program with function calls. These functions invoke the branch predictor simulator which then generates the branch prediction and updates the state of the simulated prediction hardware. By comparing the actual branch direction with the simulated predication, the performance of the branch predictor can be determined on a per branch basis. The profiler's modifications to the program do not change its behavior because the inserted functions return the actual branch conditions so that the program always executes down the correct path.

Profiles that report the performance of the Two-Level Branch Predictor for each static branch in each of the six SPECint92 benchmarks were generated. Table I lists the input data sets used to profile each of the benchmarks. The SPECint92 reference input sets were used whenever possible, but because eqntott, compress, and xlisp were each provided with

Table I. Input Data Sets used to Profile the SPECint92 Benchmarks

	Profiling Inputs		
Benchmark	Input 1	Input 2	Input 3
008.espresso	cps	bca	ti
022.li	9 queens	hanoi[a]	roots[b]
023.eqntott	int1.eqn[c]	int2.eqn[d]	fx2fp.eqn[e]
026.compress	in	gcc src[f]	trace[g]
072.sc	loada1	loada2	loada3
085.gcc	rttv.i[h]	stmt.i	gcc.i

[a] Tower of Hanoi.
[b] Newton's method for approximating square roots.
[c] Abbreviated version of the SPECint reference input set int_pri_3.eqn. It consists of 15 Boolean equations with 39 different variables.
[d] Abbreviated version of the SPECint reference input set int_pri_3.eqn. It consists of 27 Boolean equations with 49 different variables. The majority of the equations differ from those used in int1.eqn.
[e] Fixed point to floating point encoder.
[f] Concatenated gcc source files (~ 1MB).
[g] Motorola MC88110 instruction trace of compress (~ 1MB).
[h] rttv.i is the concatenation of the four SPECint92 reference input files regclass.i, toplev.i, tree.i, and varasm.i.

Table II. Percentage of Mispredictions Covered by Branches Specified as Hard-to-Predict by the *Input 1* Data Set

Benchmark	*Input 1*	Input 2	Input 3
008.espresso	*75.25*	70.55	67.28
022.li	*75.03*	68.32	50.36
023.eqntott	*83.26*	78.36	85.12
026.compress	*76.57*	83.29	86.38
072.sc	*89.66*	55.25	6.42
085.gcc	*75.01*	76.68	74.54

only one input set, their profiles were based on inputs that were not from the SPECint92 suite.

For each profile, the branches were listed from worst to best where the branch with the largest number of mispredictions was considered the worst one. Branches that appeared in the list before the cumulative number reached 75% of the total number of mispredictions were considered to be the hard-to-predict branches for that profile. Appendix A contains an abbreviated listing of these branches. The profiles show that for all the benchmarks, with the exception of gcc, there were very few hard-to-predict branches. Tables II–IV list for each benchmark the percentage of total mispredictions that were covered by the set of branches that are profiled as hard-to-predict. Each table uses a different input file to generate the profile. With the exception of sc, an average of 73% of the total mispredictions for each of the benchmarks were covered by the set of branches that were profiled as hard-to-predict. This result shows that the set of hard-to-predict branches for a given benchmark is consistent over all the input data sets profiled, indicating that the majority of the hard-to-predict branches can be detected by profiling. Sc, the spreadsheet program, did not fare as well

Table III. Percentage of Mispredictions Covered by Branches Specified as Hard-to-Predict by the *Input 2* Data Set

Benchmark	Input 1	*Input 2*	Input 3
008.espresso	35.77	*75.05*	30.57
022.li	79.52	*75.62*	59.54
023.eqntott	83.26	*78.36*	85.12
026.compress	76.57	*83.29*	86.38
072.sc	53.34	*76.34*	5.59
085.gcc	71.74	*75.02*	71.21

Table IV. Percentage of Mispredictions Covered by Branches Specified as Hard-to-Predict by the *Input 3* Data Set

Benchmark	Input 1	Input 2	*Input 3*
008.espresso	70.46	67.02	*75.18*
022.li	71.19	74.01	*75.20*
023.eqntott	83.26	78.36	*85.12*
026.compress	76.57	83.29	*86.38*
072.sc	32.94	36.70	*76.56*
085.gcc	73.89	75.38	*75.01*

because each of the input data sets focused on a different subset of the spreadsheet commands. Our experiment also yielded an anomalous result for espresso. The profiles generated from the *Input 1* and *Input 3* data sets covered the majority of the hard-to-predict branches for the other two data sets, but the *Input 2* profile did not. This is because *Input 1* and *Input 3*'s profiles had to include a large number of static branches to reach the 75% misprediction threshold while *Input 2*'s profile required a smaller number of static branches. This smaller set was subsumed by the larger sets. As a result, *Input 2*'s profile was unable to provide good coverage of the other input data sets even though their profiles were able to provide good coverage of its hard-to-predict branches.

Appendix A also lists branches that were eliminated in our experiment. Since not all branches can be predicated (e.g., loop branches can not be predicated), one of the hard-to-predict branches in the *sc* benchmark was not eliminated. In addition, because *gcc* had a large number of hard-to-predict branches and the branch elimination for each benchmark was done by hand, only a small fraction of the hard-to-predict branches was eliminated for the *gcc* benchmark.

3.3. Source Level Transformations

Once the hard-to-predict branches for a program have been found, the performance benefit provided by predicated execution is dependent on the number of these branches that can be eliminated. Branches that cannot be eliminated include loop branches and branches that branch around procedure calls. To deal with these branches, a set of transformations was proposed by Mahlke(6, 9) that transformed some of these branches into a form more amenable to elimination. Two of these transformations, loop peeling and loop branch coalescing, were used to help eliminate some of the hard-to-predict branches found in Section 3.2.

Loop peeling is applied to loop branches that are hard-to-predict because the loop frequently iterates a small number of times. The body of the loop is duplicated an appropriate number of times and the execution of each copy is predicated on the condition that the previous copy was not the last iteration of the loop. As a result, no branches need to be executed for most occurrences of the loop.

Loop branch coalescing merges separate loop-exit branches within a loop body into one branch. Loop-exit branches branch to either some point in the program outside the loop or to the basic block that follows it inside the loop body. Because these branches can redirect the instruction stream to a point inside or outside of the loop, they cannot be eliminated. For a loop with *n* loop-exit branches, loop branch coalescing reduces the number of branches that cannot be eliminated from *n* to one.

3.4. Instruction Promotion

One cost of using predicated execution to eliminate a branch is the introduction of new data dependencies. An instruction that was converted into a predicated instruction becomes data dependent on the instruction that generates the branch's condition. Instruction promotion[6, 16] can remove this dependence by promoting the instruction above the branch so that it does not need to be predicated. This optimization can only be applied when the destination register of the promoted instruction is dead

```
;; Predicated assembly code
;;
ld      r4, r6, 16
cmp_eq  r5, r6,  0
ld      r8, r6,  0  if r5
add     r2, r2, r8  if r5


;; Predicated assembly code
;;   with instruction promotion.
;;
ld      r4, r6, 16
ld      r8, r6,  0       ; Instruction promoted
cmp_eq  r5, r6,  0
add     r2, r2, r8  if r5
```

Fig. 3. Example of instruction promotion which removes the data dependence on the predicate.

at the point of the branch (i.e., along both paths of the branch, the register is written before its first use) and that the promoted instruction is the first instruction to write to the register along its branch path. Figure 3 gives an example of instruction promotion from the sc benchmark. When the promoted instruction can potentially cause an exception (e.g., a load instruction), a nontrapping version of the instruction is used.(22, 23)

4. EXPERIMENTAL RESULTS

4.1. Generating the Predicated Object Files

The code compilation process consists of the following steps. First, we profile the benchmarks to identify the hard-to-predict branches. We then hand-modify the C source-code programs to include the aforementioned source level transformations, converting selected branches into short if-branches. The assembly code for the modified source programs are generated using the GCC V2.4.3 compiler. We then hand-modify the assembly programs, predicating the hard-to-predict branches that can be eliminated and performing the aforementioned instruction promotion optimizations. [The manual steps in the compilation process for our experiments can be automated as is done in the IMPACT compiler.(6, 9) We are currently working on incorporating those steps into our compiler.] Finally, predicated object files are generated using the GNU assembler. For the gcc benchmark, profiling showed many hard-to-predict branches. Because we are hand-modifying the source-code program, only a small subset of those hard-to-predict branches were eliminated. Appendix A lists for each benchmark the branches that were actually eliminated.

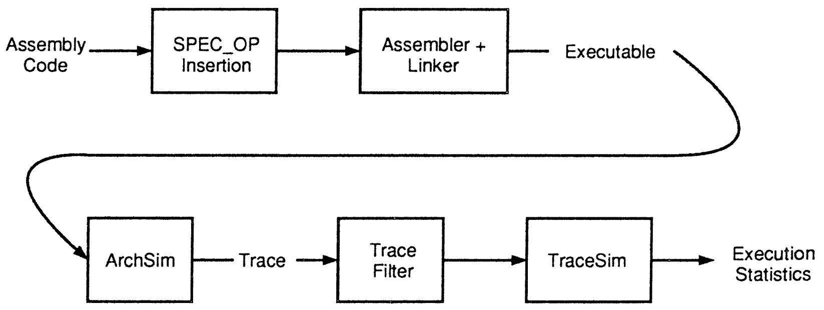

Fig. 4. Process flow for simulating predicated instructions.

Before	After
⋮	⋮
beq r0, 0, L1	SPEC_OP
op1	beq r0, 0, L1
op2	op1
L1: ···	op2
	L1: ···

Fig. 5. Marking branches in the assembly code for predication with the special instruction SPEC_OP.

4.2. Simulation Process

The simulation process consists of the following steps (see Fig. 4). First, we insert special instructions, SPEC_OPs, in the assembly program to indicate which branches are to be predicated. Figure 5 shows a code segment before and after a branch has been marked for predication. This modification to the program does not change its behavior because these special instructions are NOPs and do not affect the machine state. A Motorola MC88100 instruction level simulator, ArchSim, then reads in the object code and simulates execution, producing an instruction trace. To replace the selected branches with predicated instructions, a trace-filter module reads the trace generated by ArchSim and scans for occurrences of the SPEC_OP instruction. If a SPEC_OP instruction is detected, the filter module does the following:

- If the branch following the SPEC_OP instruction is not taken, we replace the instructions in the fall-through path with predicated instructions (see Fig. 6).

Before	After
⋮	⋮
SPEC_OP	op1 if r0
beq r0, 0, L1	op2 if r0
op1	L1: ···
op2	
L1: ···	

Fig. 6. Replacing a not-taken branch in the instruction trace with the appropriate set of predicate instructions.

- If that branch is taken, the instructions in the fall-through path will not be in the instruction trace. To determine what predicated instructions to insert into the new instruction stream, the filter module reads in a table which contains the predicated instructions associated with each branch (see Fig. 7).

The new instruction trace is then processed by the trace-driven simulator, TraceSim, to produce the execution statistics.

4.3. Machine Model

The underlying microarchitectural model is an HPS implementation[1, 19] of the MC88100 architecture. Execution in HPS flows as follows: Each cycle, multiple instructions are issued, and using the information in the register files, the instructions are merged into node tables, much like the Tomasulo algorithm merges operations into the reservation stations of the IBM 360/91.[18] Associated with each instruction (node) are the source operands for that instruction (or identifiers for obtaining the operands), and destination information. Each node is stored in its proper node table independent of and decoupled from all other nodes currently awaiting dependencies in the datapath until all its operands are available, at which point the node is eligible for scheduling. Each cycle, the oldest firable node of each node table is scheduled, i.e., it is shipped to a pipelined functional unit for execution. Each cycle, functional units complete execution of nodes and distribute the results to nodes waiting for these results, which then may become firable. Many of these microarchitectural features which comprise the HPS model have been adopted by industry and incorporated into their current microprocessor implementations.[24, 25]

The HPS processor simulated in this paper supports 8 wide issue with a perfect instruction cache and a 16KB data cache. The data cache miss latency is 10 cycles. Table V shows the instruction classes and their simulated execution latencies, along with a description of the instructions

Before		After	
	⋮		⋮
	`SPEC_OP`		`op1 if r0`
	`beq r0, 0, L1`		`op2 if r0`
`L1:`	`...`	`L1:`	`...`

Fig. 7. Replacing a taken branch in the instruction trace with the appropriate set of predicate instructions.

Table V. Instruction Classes and Latencies

Instruction Class	Exec. Lat.	Description
Integer	1	INT add, sub and logic OPs
FP Add	3	FP add, sub, and convert
FP/INT Mul	3	FP mul and INT mul
FP/INT Div	8	FP div and INT div
Load	2	Memory loads
Store	—	Memory stores
Bit Field	1	Shift, and bit testing
Branch	1	Control instructions

that belong to each class. In the processor simulated, each functional unit can execute instructions from any of the instruction classes. The maximum number of instructions that can exist in the machine at one time is 128. An instruction is considered in the machine from the time it is issued until it is retired.

4.4. Experiments

Experiments were run to measure the performance benefit of adding predicated execution for the four SPECint92 benchmarks, eqntott, compress, sc, and gcc. Four different variations for each benchmark were simulated:

- *np*—baseline version of the benchmark in which none of the branches were eliminated.
- *sp*—software-based predication version in which branches were eliminated by the GCC compiler through the use of logical and bit-manipulation operations.(26) Appendix B shows an example of software-based predication.
- *hp*—ISA-based predication version in which branches were eliminated by predicated instructions.
- *ip*—ISA-based predication version in which branches were eliminated by predicated instructions and instruction promotion optimizations were applied.

The branches that were considered for elimination by predicated execution were chosen based on the profiles of the *Input* 1 data set for each benchmark (see Section 3.2). Appendix A lists the subset of those branches that were actually eliminated in the *hp* and *sp* versions. For the sc and gcc benchmarks, the compiler did not eliminate any branches through software predication. The *sp* and *np* versions were identical for those benchmarks. Experimental runs were done using the *Input 1*, *Input 2*, and *Input 3* data sets for each benchmark. Although the *Input 1* results were unrealistically optimistic because they were based on the use of the same data set for both profiling and execution, they were included to provide a baseline to which the *Input 2* and *Input 3* results could be compared.

Figures 8–11 show the absolute number of mispredictions for each benchmark. Because the number of mispredictions for the *hp* and *ip* versions were the same, the numbers for the *ip* version were omitted.

The ISA-predicated versions of the compress and eqntott benchmarks showed large reductions in absolute misprediction counts across all three input data sets. The relative drops in the number of mispredictions for the second and third data sets were consistent with the first data set, indicating that profiling was locating a significant number of the hard-to-predict branches. In addition, the misprediction count for compress's *hp* version was significantly lower than the misprediction count for its *sp* version because ISA-based predication was able to eliminate more of the hard-to-predict branches than software-based predication. The eqntott benchmark showed little difference between its *hp* and *sp* versions because software-based predication was able to eliminate all the hard-to-predict branches eliminated by ISA-based predication.

The ISA-predicated versions of the gcc benchmark showed small reductions in mispredictions across all three input data sets. Because gcc had a large number of hard-to-predict branches and the branch elimination for each benchmark was done by hand, only a small fraction of the hard-to-predict branches were eliminated for the gcc executable used in this study. Only the top ten hard-to-predict branches were considered for elimination. These branches accounted for 7.8% of the total mispredictions in the profile run. By considering only these branches, we were able to reduce the total number of mispredictions for the three input sets by 3.5% on the average. Predication's full performance benefit will not be known until the process is automated and we are able to consider all branches in the benchmark. However, if the branches we considered are representative of the rest of gcc's hard to predict branches, then predicated execution could reduce gcc's mispredictions by 40%.

The ISA-predicated versions of the sc benchmark showed a large reduction in the absolute misprediction count for the first input data set, but little reduction in the misprediction counts for the second and third input sets. Given that the first input data set was used to profile the benchmark, this result indicates that while predicated execution can be effective in eliminating the hard-to-predict branches in the sc benchmark, profiling was not effective in locating them.

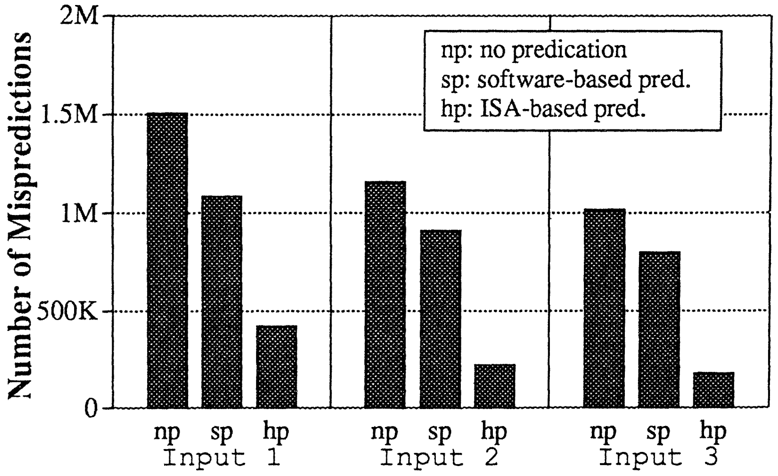

Fig. 8. Predicated execution's effect on the dynamic number of mispredictions—compress.

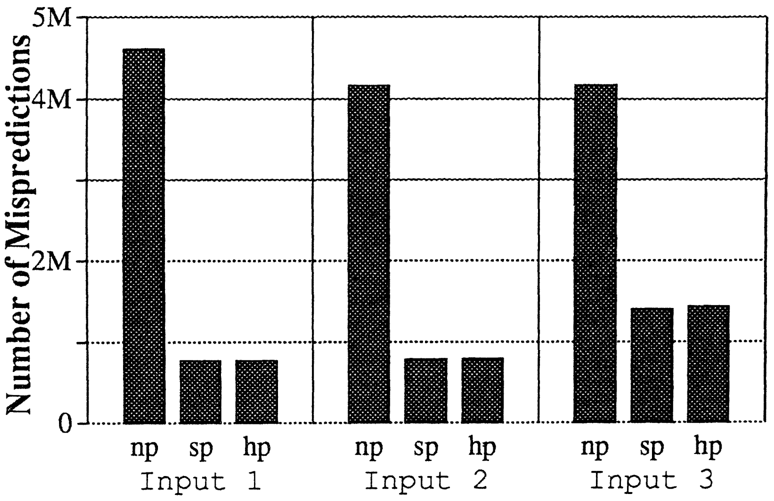

Fig. 9. Predicated execution's effect on the dynamic number of mispredictions—eqntott.

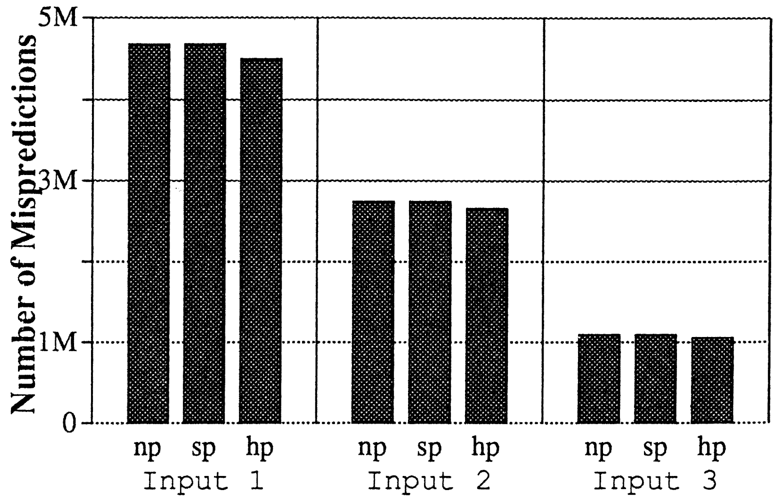

Fig. 10. Predicated execution's effect on the dynamic number of mispredictions—gcc.

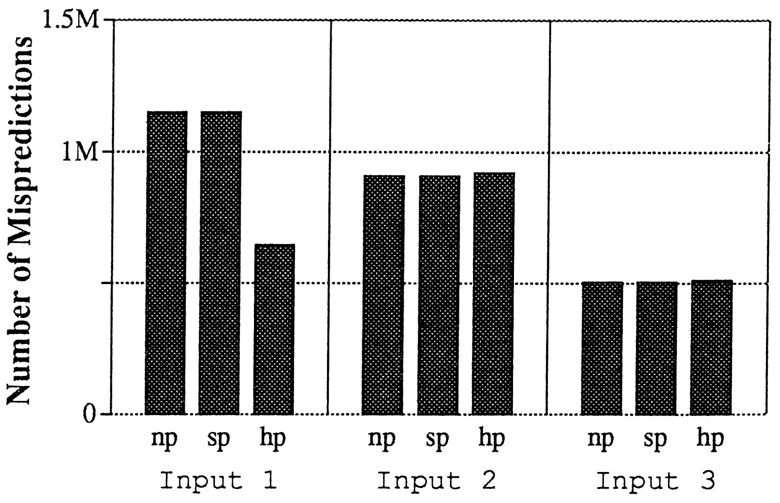

Fig. 11. Predicated execution's effect on the dynamic number of mispredictions—sc.

Figures 12–15 show the total number of cycles needed to execute each of the benchmarks. This number was broken down into two components, the total number of cycles spent on the correct path of the program (i.e., doing the real work) and the total number of cycles that were spent on incorrect paths of the program, waiting for mispredicted branches to be resolved.

The execution times of the *hp* and *ip* versions of the compress benchmark were 23% and 25% faster than that of the *np* version. This speedup was due solely to the reduction in cycles wasted due to branch misprediction. The *hp* and *ip* versions were actually spending additional time executing along the correct path. This effect was due to the predicated instructions increasing the latency of the program because of their extra flow dependencies. Some of this latency was eliminated in the *ip* version by the instruction promotion optimization. Its running time was 3% faster than that of the *hp* version. Despite its smaller number of mispredictions, the *sp* version was slower than the *np* version by an average of 9.7%. The slowdown was caused by the software predication. Elimination of an inner-loop branch in the *sp* version required a large number of logical operations that greatly increased the critical path within the loop. This resulted in an

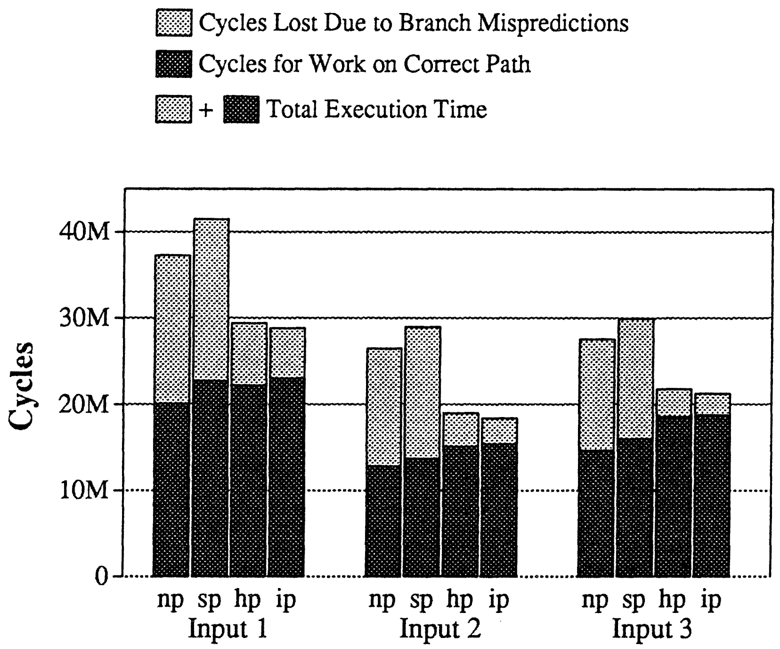

Fig. 12. Predicated execution's effect on execution time—compress.

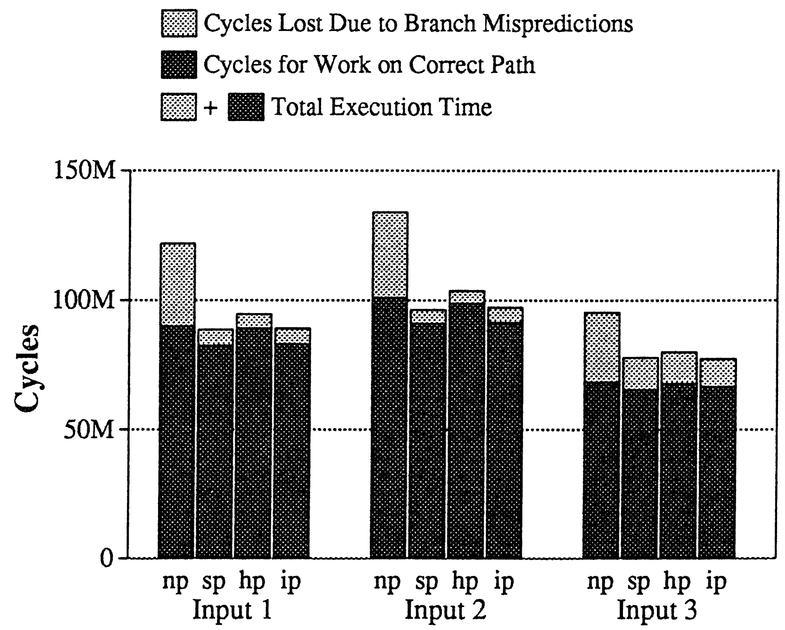

Fig. 13. Predicated execution's effect on execution time—eqntott.

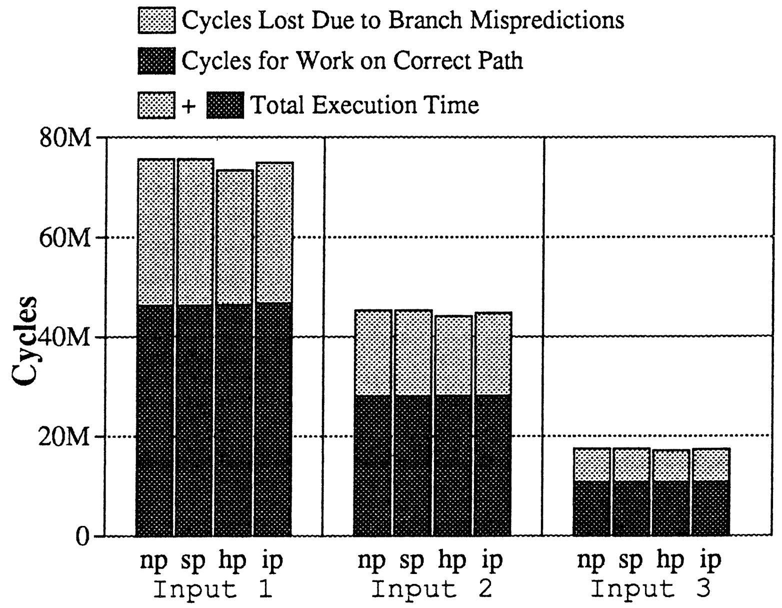

Fig. 14. Predicated execution's effect on execution time—gcc.

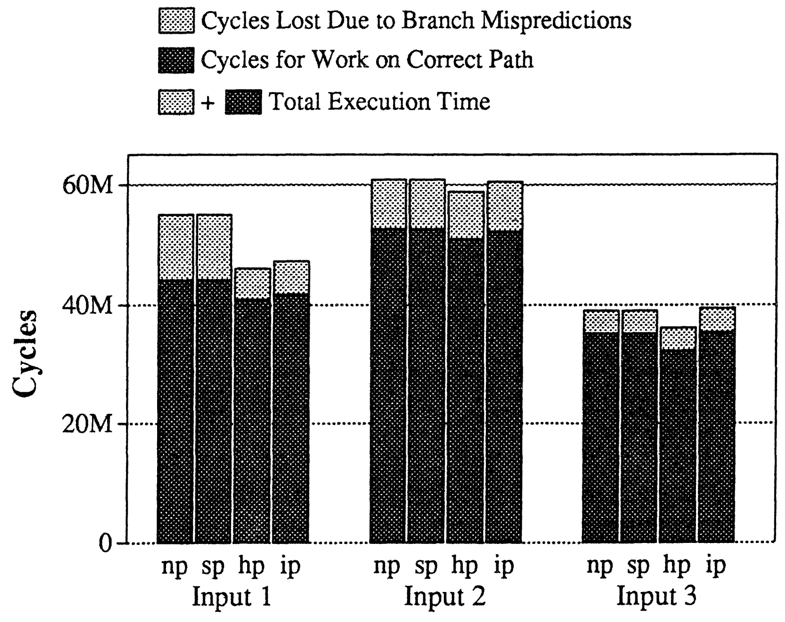

Fig. 15. Predicated execution's effect on execution time—sc.

increase in execution time that could not be offset by the number of cycles saved due to the elimination of the branch.

The *hp* and *ip* versions of the eqntott benchmark outperformed the *np* version by 20% and 24%, respectively. For the *hp* version, this increase in performance was almost entirely due to the difference in branch execution penalty. The *ip* version was able to achieve an additional increase of 4% by reducing the number of cycles required to execute the real work of the program. In fact, this number was even smaller than that of the baseline *np* version. This reduction in the number of cycles required for doing the real work of the program was due to the increase in basic block size and issue density that occurs when branches are eliminated by predicated execution. This performance increase was not apparent in the *hp* version because it did not take advantage of the instruction promotion optimization to reduce the latency penalties of its predicated instructions. Unlike compress's *sp* version, eqntott's *sp* version showed a significant performance increase over the *np* version (24%) because the sequences of logical operations required to eliminate the hard-to-predict branches were extremely short.

The *hp* and *ip* predicated versions of the gcc benchmark outperformed the *np* version by a very small margin, 2.5% and 1.0%. As mentioned above, the difference in performance was small because this study attempted to eliminate only a small fraction of the hard-to-predict branches.

The *hp* and *ip* predicated versions of the sc benchmark showed small performance increases over the *np* version for the second and third input sets. The sc benchmark did not achieve the same level of improvement as the other benchmarks because the profile generated by its first input set did not locate many of the hard-to-predict branches for its second and third input sets.

Unlike the other benchmarks, sc's *hp* version outperformed its *ip* version. Because the instruction promotion optimization removes flow dependencies from the program, it is counter-intuitive that the use of this optimization would reduce performance. The predicated instruction model used in this experiment provides a possible explanation to this anomaly. A predicated instruction whose predicate evaluates to false does not have to wait for its source operands. In the case of a store instruction, this means throwing away the store as soon as the predicate evaluates to false, this is, without waiting for the target address to be resolved. This allows the dynamic memory disambiguator to disregard this instruction. On the other hand, when the store instruction is promoted, it cannot be executed until its target address has been resolved. Furthermore, the memory disambiguator forces all memory instructions that follow the store to also wait for the resolution of the target address. If the target address is slow to resolve, it will delay the execution of its associated store instruction and all subsequent memory instructions. This situation occurs in the sc benchmark where the majority of the promoted instructions were store and load instructions that were inside loops.

5. CONCLUSION

This study examined the performance benefit of combining speculative and predicated execution to reduce branch execution penalty. Speculative execution eliminates the branch execution penalty for branches that are predicted correctly. Predicated execution can eliminate a branch's execution penalty regardless of the processor's ability to predict it correctly. However, it incurs a small performance overhead and is not applicable to all branches. To achieve a better combination of these two mechanisms, this study used predicated execution to handle the hard-to-predict branches and speculative execution to handle the remaining branches. Profiling was used to determine the hard-to-predict branches for each benchmark. The performance benefit of this approach was measured for a wide-issue dynamically scheduled processor. In addition, this study examined the effect on performance of the instruction promotion compiler optimization.

The results show that profiling is an effective mechanism for detecting hard-to-predict branches. For a given benchmark, one input data set was chosen to generate the profile. The effectiveness of the profile was then measured for all the input data sets for the benchmark. With the exception of sc, the set of branches denoted by profiling as hard-to-predict accounted for an average of 73% of the total mispredictions for each of the SPECint92 benchmarks.

By using ISA-based predication to eliminate only the branches from the profile-generated set of hard-to-predict branches, significant performance improvements were achieved for compress and eqntott (23% and 20%). Profiling was effective in locating the hard-to-predict branches for both these benchmarks and predicated execution was effective in eliminating those branches. Predicated execution provided only a small performance benefit for gcc (2.5%). However this result was considered promising because the limitations of this study forced us to consider only a very small subset of gcc's hard-to-predict branches for elimination. Based on this result, we project that predicated execution could reduce gcc's mispredictions by 40%. The predicated version for sc showed little improvement in performance. Because profiling was not effective in finding the hard-to-predict branches for sc, predicated execution was not able provide much performance benefit.

Software-based predication can also significantly improve performance. The eqntott benchmark showed little difference between its *hp* and *sp* versions because software-based predication was able to efficiently eliminate all the hard-to-predict branches eliminated by ISA-based predication. However, software-based prediction can also degrade performance when not carefully applied. The *sp* version of the *compress* benchmark was slower than the baseline model because software-based predication greatly increased the critical path of a program after eliminating a hard-to-predict branch with a large number of logical operations. This increase in execution time was not offset by the number of cycles saved due to the elimination of the branch.

The performance benefit of using the instruction promotion optimization was not consistent across the benchmarks. The compress and eqntott benchmarks showed a small increase in performance (3%–5%) due to instruction promotions. The sc benchmark showed a decrease in performance (up to 9%). This decrease in performance was due to the promotion of certain memory instructions that hindered the dynamic memory disambiguator. When the memory instructions were predicated, a significant number of them were squashed, allowing subsequent memory instructions to resolve earlier. This suggests that the instruction promotion optimization must be selectively applied in order to achieve performance improvements.

This study's results showed that adding predicated execution to a machine that supports speculative execution can lead to significant increases in performance. However, more research can still be done to further increase the performance benefit provided by predicated execution. In particular, predicated execution's performance benefit would be significantly increased by compiler optimizations that transform the code so that more of the hard-to-predict branches could be eliminated. We are currently building a compiler to not only automate branch elimination through predication, but to evaluate the performance benefits of these optimizations as well.

ACKNOWLEDGMENTS

This paper is one result of our ongoing research in high performance computer implementation at the University of Michigan. The support of our industrial partners, Intel, NCR in all its various incarnations, Motorola, and Hewlett–Packard, is greatly appreciated. In addition, we wish to gratefully acknowledge the other members of the HPS research group for the stimulating environment they provide and for their comments and suggestions on this work. We are particularly grateful to NCR and Intel for their technical and financial support, and to Hewlett-Packard for the gift of the HP 7xx workstations. We would also like to thank the reviewers of this paper for their helpful suggestions.

APPENDIX A

A.1. Hard-to-Predict Branches

The tables in this section list up to the ten worst branches in descending order for each benchmark simulated. The branch which accounts for the largest number of branch mispredictions is considered to be the worst. A check in the *hp* or the *sp* column indicates that the branch is removed in the *hp* or *sp* version of the program respectively.

026.compress:

	Line	% misprediction				
File	No.	input1	input2	input3	*hp*	*sp*
compress.c	790	29.06	35.50	41.37	√	
compress.c	799	27.31	19.16	20.07	√	√
compress.c	802	20.20	28.63	24.94	√	
	total :	76.57	83.29	86.38		

023.eqntott:

File	Line No.	% misprediction			*hp*	*sp*
		input1	input2	input3		
pterm_ops.c	45	58.52	52.40	59.71	✓	✓
pterm_ops.c	47	24.74	25.96	25.41	✓	✓
	total :	83.26	78.36	85.12		

085.gcc:

File	Line No.	% misprediction			*hp*	*sp*
		input1	input2	input3		
tree.c	408	2.31	2.13	1.84	✓	
c-parse.tab.c	1016	2.02	2.15	1.56		
c-parse.tab.c	3185	1.57	1.62	1.11	✓	
reload.c	1190	1.52	1.25	1.72		
c-parse.tab.c	3185	1.45	1.54	0.99	✓	
c-parse.tab.c	1056	1.41	1.53	1.13	✓	
c-parse.tab.c	2272	1.29	1.25	1.09	✓	
tree.c	522	1.23	1.40	0.82	✓	
c-parse.tab.c	1056	1.15	1.16	0.92	✓	
reload.c	1190	1.14	1.02	1.27		
	total :	15.09	15.05	12.45		

072.sc:

File	Line No.	% misprediction			*hp*	*sp*
		input1	input2	input3		
interp.c	264	36.32	0.00	0.85	✓	
interp.c	1002	32.11	27.55	5.18	✓	
interp.c	1001	21.22	27.70	0.40		
	total :	89.66	55.25	6.03		

APPENDIX B

B.1. ISA-Based and Software-Based Predication

This section gives an example of a branch being eliminated by ISA-based predication and being eliminated by software-based predication.

- No predication

```
   I1: cmpeq r4, r2, r3
   I2: bne   r4, 1, L1
   I3: mov   r10, r8
   I4: jmp   L2
L1:
   I5: mov   r10, r9
L2:
```

- ISA-based predication

```
I1: cmpeq r4, r2, r3
I2: mov   r10, r8     if r4
I3: mov   r10, r9     ifnot r4
```

- Software-based predication

```
I1: cmpeq r4, r2, r3
I2: ext   r6, r4, 1<0>          /* if r4's 0th bit == 1, r6 = 0xffffffff */
                                /* otherwise, r6 = 0x0 */
I3: and   r8, r8, r6

I4: cmpne r5, r2, r3
I5: ext   r7, r5, 1<0>
I6: and   r9, r9, r7

I7: or    r10, r8, r9
```

REFERENCES

1. Y. N. Patt, S. W. Melvin, W. Hwu, and M. C. Shebanow, Critical issues regarding HPS, a high performance microarchitecture, *Proc. of the 18th Ann. ACM/IEEE Int'l. Symp. on Microarchitecture*, pp. 109–116 (1985).
2. S. Melvin and Y. N. Patt, Exploiting fine-grained parallelism through a combination of hardware and software techniques, *Proc. of the 18th Ann. Int'l. Symp. on Computer Architecture*, pp. 287–297 (1991).
3. P. Hsu and E. Davidson, Highly concurrent scalar processing, *Proc. of the 13th Ann. Int'l. Symp. on Computer Architecture* (1986).
4. B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle, The Cydra 5 departmental supercomputer, *IEEE Computer*, **22**:12–35 (January 1989).
5. J. C. Dehnert, P. Y. T. Hsu, and J. P. Bratt, Overlapped loop support in the Cydra 5, *Proc. of the 16th Ann. Int'l. Symp. on Computer Architecture*, pp. 26–38 (1989).
6. S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, Effective compiler support for predicated execution using the hyperblock, *Proc. of the 25th Ann. ACM/IEEE Int'l. Symp. on Microarchitecture*, pp. 45–54 (1992).
7. D. N. Pnevmatikatos and G. S. Sohi, Guarded execution and dynamic branch prediction in dynamic ILP processors, *Proc. of the 21st Ann. Int'l. Symp. on Computer Architecture*, pp. 120–129 (1994).
8. G. S. Tyson, The effects of predication on branch prediction, *Proc. of the 27th Ann. ACM/IEEE Int'l. Symp. on Microarchitecture*, pp. 196–206 (1994).
9. S. A. Mahlke, R. E. Hank, R. A. Bringmann, J. C. Gyllenhaal, D. M. Gallagher, and W. W. Hwu, Characterizing the impact of predicated execution on branch prediction, *Proc. of the 27th Ann. ACM/IEEE Int'l. Symp. on Microarchitecture*, pp. 217–227 (1994).
10. E. M. Riseman and C. C. Foster, The inhibition of potential parallelism by conditional jumps, *IEEE Trans. on Computers*, **C-21** (12):1405–1411 (1972).
11. J K. F. Lee and A. J. Smith, Branch prediction strategies and branch target buffer design, *IEEE Computer*, pp. 6–22 (January 1984).
12. J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, Conversion of control dependence to data dependence, *10th Ann. ACM Symp. on Principles of Programming Languages*, pp. 177–189 (1983).

13. T.-Y Yeh and Y. N. Patt, Two-level adaptive branch prediction, *Proc. of the 24th Ann. ACM/IEEE Int'l. Symp. on Microarchitecture*, pp. 51–61 (1991).
14. T.-Y. Yeh and Y. N. Patt, Alternative implementations of two-level adaptive branch prediction, *Proc. of the 19th Ann. Int'l. Symp. on Computer Architecture*, pp. 124–134 (1992).
15. P.-Y. Chang, E. Hao, T.-Y. Yeh, and Y. N. Patt, Branch classification: A new mechanism for improving branch predictor performance, *Proc. of the 27th Ann. ACM/IEEE Int'l. Symp. on Microarchitecture*, pp. 22–31 (1994).
16. P. Tirumalai, M. Lee, and M. Schlanskar, Parallelization of loops with exits on pipelined architectures, *Proc. Supercomputing* '90, (1990).
17. M. G. Butler, Aggressive execution engines for surpassing single basic execution, Ph.D. thesis, University of Michigan, 1993.
18. R. M. Tomasulo, An efficient algorithm for exploiting multiple arithmetic units, *IBM Journal of Res. and Development*, **11**:25–33 (January 1967).
19. Y. Patt, W. Hwu, and M. Shebanow, HPS, a new microarchitecture: Rationale and introduction, *Proc. of the 18th Ann. ACM/IEEE Int'l. Symp. on Microarchitecture*, pp. 103–107 (1985).
20. E. Sprangle and Y. Patt, Facilitating superscalar processing via a combined static/dynamic register renaming scheme, *Proc. of the 27th Ann. ACM/IEEE Int'l. Symp. Microarchitecture*, pp. 143–147 (1994).
21. S. McFarling, Combining branch predictors, Technical Report TN-36, Digital Western Research Laboratory (June 1993).
22. M. D. Smith, M. S. Lam, and M. A. Horowitz, Boosting beyond static scheduling in a superscalar processor, *Proc. of the 17th Ann. Int'l. Symp. on Computer Architecture*, pp. 344–354 (1990).
23. P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, IMPACT: An architectural framework for multiple-instruction-issue processors, *Proc. of the 18th Ann. Int'l. Symp. on Computer Architecture*, pp. 266–275 (1991).
24. L. Gwennap, Intel's P6 uses decoupled superscalar design, *Microprocessor Report*, Vol. 9, (February 1995).
25. L. Gwennap, PA-8000 combines complexity and speed, *Microprocessor Report*, Vol. 8, No. 15 (November 1994).
26. T. Granlund and R. Kenner, Eliminating branches using a superoptimizer and the GNU C compiler, *Proc. of the ACM SIGPLAN '92 Conf. on Programming Language Design and Implementation*, pp. 341–352 (1992).