Architectural Support for Compiler-Synthesized Dynamic Branch Prediction Strategies: Rationale and Initial Results

David I. August Daniel A. Connors John C. Gyllenhaal Wen-mei W. Hwu
Center for Reliable and High-Performance Computing
University of Illinois
Urbana-Champaign, IL 61801
Email: {august, dconnors, gyllen, hwu}@crhc.uiuc.edu

Abstract

This paper introduces a new architectural approach that supports compiler-synthesized dynamic branch predication. In compiler-synthesized dynamic branch prediction, the compiler generates code sequences that, when executed, digest relevant state information and execution statistics into a condition bit, or predicate. The hardware then utilizes this information to make predictions. Two categories of such architectures are proposed and evaluated. In Predicate Only Prediction (POP), the hardware simply uses the condition generated by the code sequence as a prediction. In Predicate Enhanced Prediction (PEP), the hardware uses the generated condition to enhance the accuracy of conventional branch prediction hardware.

The IMPACT compiler currently provides a minimal level of compiler support for the proposed approach. Experiments based on current predicated code show that the proposed predictors achieve better performance than conventional branch predictors. Furthermore, they enable future compiler techniques which have the potential to achieve extremely high branch prediction accuracies. Several such compiler techniques are proposed in this paper.

1 Introduction

State-of-the-art dynamic branch prediction schemes use hardware mechanisms that transform branch execution histories into predictions. These schemes record branch execution history in the form of counters or explicit bit patterns. They use this record of past behavioral information to accurately predict future behavior. However, little effort has been made to explicitly take advantage of any other relevant program state information to further enhance the accuracy of these predictions.

This paper introduces a new architectural approach that taps into other relevant program state information. The proposed approach consists of three parts: branches based on explicit condition bits at the architecture level, a prediction mechanism based on explicit condition bits at the microarchitecture level, and code generation support at the compiler level. The use of explicit condition bits, referred to as predicates in this paper, at all three levels provide a simple yet powerful interface to support sophisticated run-time branch prediction schemes.

At the architectural level, a set of branch instructions are defined which base their decision on a predicate. These branch instructions are similar to those discussed in [1] and defined in the IBM RS6000 [2], Cydrome Cydra-5 [3], HPL PlayDoh [4], and SPARC V9 [5]. Each branch requires a previously executed instruction to set a predicate. Therefore, more instructions are potentially required in the compare-and-branch model, such as in the HP PA-RISC architecture [6]. However, in future architectures that support predicated execution [4][7], the cost of these predicate defining instructions is often mitigated by the reduction of branches.

At the microarchitecture level, two branch prediction mechanisms are presented which support the proposed approach. In Predicate Only Prediction (POP), the hardware simply uses the branch predicate value as a prediction. The branch predictor simply accesses the predicate register used by the branch instruction. If the predicate value is available, no prediction is required and a perfect decision is made early for the branch. If the predicate does not yet contain the correct value for the branch, the current value is used for prediction, rather than stalling. The use of a previous version of the predicate value allows the compiler to influence the branch predictor by assigning values to this version of the predicate.

In Predicate Enhanced Prediction (PEP), the hardware uses the predicate value to enhance the accuracy of conventional branch prediction hardware. In the proposed PEP schemes, the predicate value is used to steer the update and use of the branch history information. In cases where the predicate provides useful information, such steering allows the compiler-synthesized prediction to achieve higher accuracy. In cases where the predicate does not offer useful information, the proposed PEP schemes simply degenerate to the conventional branch prediction hardware that they were built upon.

The conditions in the predicates used by both the POP and PEP schemes are determined when the program is executed. Ideally, the compiler synthesizes code sequences that set the predicates with relevant information. However, as we will illustrate in this paper, such code sequences often exist naturally in the current generation of predicated code. This phenomenon makes the proposed scheme attractive since code generated by existing predication compilers immediately benefits from them. However, existing

Stage	Instruction
Fetch	Branch p1, DEST
Decode	p1 = Condition
Execute	
Write Back	

(a)

Stage	Instruction
Fetch	Branch p1, DEST
Decode	
Execute	
Write Back	p1 = Condition

(b)

Figure 1: Timing of code segments flowing through a pipelined processor. In (a), the branch condition has not been computed before branch direction is needed. In (b), the condition has been computed before branch direction is needed.

code does not take full advantage of the potential of the proposed approach. This paper also proposes some practical methods by which a compiler could generate code to provide relevant branch information.

The rest of this paper illustrates the potential of predicate-based compiler-synthesized dynamic branch prediction. Section 2 provides the intuitive rationale behind the proposed scheme in a pipelined processor. Section 3 presents an overview of the architecture necessary for a set of predicate-based branch predictors. The Predicate Only Prediction (POP) is first presented. Then, a set of Predicate Enhanced Prediction (PEP) strategies are discussed. Section 4 discusses compiler opportunities for a both schemes. Some initial performance results of these predicate-based branch prediction schemes are shown in Section 5. Finally, the paper concludes with a discussion of future work.

2 Rationale

This section briefly presents some intuitive rationale behind predicate-based compiler-synthesized dynamic branch prediction. This discussion will be expanded upon in Section 4, after the specific schemes are described in Section 3.

Early-Resolved Branches Figure 1 shows two branches, each with its own separate condition evaluation instruction. The condition evaluation instruction computes the branch condition and places the result into a predicate register p1. If p1 is TRUE, the branch directs the flow of control to DEST. If p1 is FALSE, the branch allows the flow of control to fall through.

Figure 1(a) shows one scenario of code flowing through a pipelined processor. The figure depicts the pipeline state when a prediction on the branch needs to be made. Even though the compare instruction is fetched one stage before the branch, the execution of the compare instruction is not completed before the branch exits the fetch stage. Thus, the direction of the branch is not known in time to fetch the appropriate subsequent instructions. In other words, the branch condition is resolved late.

A similar situation is shown in Figure 1(b). The difference here is that the compare instruction is fetched three stages before the branch. This has an interesting effect on the state of the pipeline when the branch prediction is normally made. Since the compare instruction is many stages ahead of the branch, it has already been executed when the calculation of the next instruction address after the branch must occur. Since the branch condition is already determined, a prediction is unnecessary. If the branch architecture could take advantage of this fact, it would be able to make a perfect prediction. Any architecture with multiple condition codes or predicate registers could be made to take advantage of this situation.

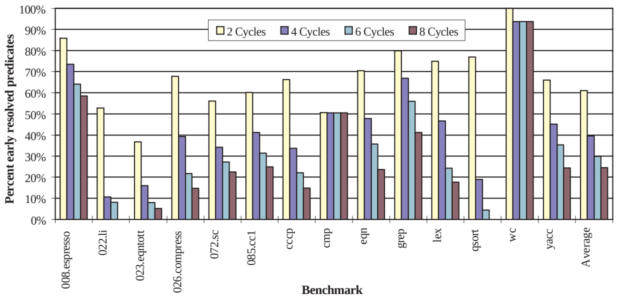
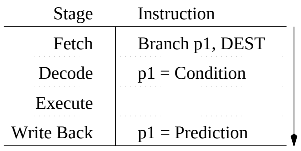


Figure 2: The percentage of early-resolved branches encountered dynamically with respect to number of cycles before the branch that the branch condition is computed. The code is generated for a 4-issue machine with support for predicated execution.

As a general rule, a branch's direction can be known if the distance between the branch and the comparison in the schedule is larger than the number of pipe stages between fetch and execute. A branch which satisfies this condition is referred to as *early-resolved*. The percentage of early-resolved branches encountered dynamically with code compiled for a 4-issue processor is shown in Figure 2. Discussion of the compiler and architecture assumptions used to generate this code will be given in Section 5. It is interesting to note that a significant portion of the branches remain early-resolved even in long pipelines. Many of these early-resolved branches are a side-effect of the if-conversion process as explained in Section 4. These early-resolved branches help make predicate-based branch predictors extremely attractive.

Correlation to Program State In general, program dependencies prevent the compiler from always computing branch conditions early enough to make all branches early-resolved. In these cases, it may be useful for the compiler to generate code to evaluate another condition which can be scheduled early enough to be used by the branch hardware. This new condition could be used in forming the branch's prediction. Figure 3 illustrates this case. Notice that the original condition must remain to correctly determine the actual direction of the branch.

At compile time, relations and correlations between branches to be predicted and other architectural state may be determined. For example, the behavior of a branch may be correlated to the control flow path which was traversed to reach it. Alternatively, a branch may be found to be highly correlated to certain values in integer or floating



Stage	Instruction
Fetch	Branch p1, DEST
Decode	p1 = Condition
Execute	
Write Back	p1 = Prediction

Figure 3: Pipeline timing of a code segment in which the branch condition has not been computed before branch direction is needed, but the compiler has inserted an early prediction.

point registers. Profile-based compilation can be used as a tool for extracting correlations [8]. With this information, the compiler can generate expressions that, when evaluated at run-time, are useful in generating accurate predictions.

Flexibility An interesting advantage that a compiler-synthesized dynamic branch prediction approach may have over other schemes is the ability to improve performance long after a processor design has been frozen. Compilers can adapt their prediction schemes to match new and different workloads. Implementing new and more sophisticated branch predictors becomes a matter of enhancing the compiler. It is also very flexible in that virtually any prediction scheme that can be computed by an inlined code sequence can be employed for branch prediction.

3 Predicate-Based Branch Prediction

While the interface to a compiler-synthesized dynamic branch prediction scheme is generic, one can derive important insight into the approach by studying it in regard to a baseline architecture with predication support. The proposed schemes, known collectively as predicate-based branch predictors, will be studied in the context of this architecture.

3.1 Baseline Architecture

Predicate-based compiler-synthesized dynamic branch prediction schemes assume an architecture with support for predicated execution [9][4]. Each instruction in a machine with support for predicated execution has a predicate source operand. The execution of each instruction in such a machine is controlled by the value of its predicate.

With predicated execution, there are two ways in which instructions can be conditionally executed. The first way employs branch instructions to direct the flow of control through or around the instructions to be guarded. The selection of the branch destination is determined by some condition and a target address. In a machine with predicated execution support, it is natural, though not necessary, to replace the branch condition with a predicate.

The second way in which instructions can be conditionally executed is through the use of predicated execution. Fetched instructions may be conditionally nullified in the pipeline by setting its predicate with the appropriate condition. The values of the predicate registers are set by a collection of predicate defining instructions. Using predication, many branches can be replaced by predicate defining instructions.

The Hewlett-Packard Laboratories PlayDoh architecture is assumed as the baseline architecture [4]. The PlayDoh architecture supports predicated execution and a branch architecture which uses predicates as branch conditions. The only difference between PlayDoh and the architecture used in this paper is in its handling of branch destinations. PlayDoh splits the branch into three instructions by moving the branch target specifier into a separate instruction. This paper assumes that the branch target specifier remains with the branch instruction. While this difference affects the interpretation of the experimental results, it does not affect the applicability of the proposed methods.

3.2 Predicate Only Prediction

The simplest scheme which can take advantage of early-resolved branches and compiler-synthesized predictions is Predicate Only Prediction (POP). A conceptual design of the POP hardware is shown in Figure 4. The POP Branch Target Buffer (BTB) contains a predicate register number field in addition to the conventional BTB fields. At the first misprediction of a branch, an entry is created in the POP BTB. Creation of the entry involves recording the predicate register number which the branch uses as its condition. Any time the branch is fetched, a lookup in the POP BTB finds the previously created entry. The predicate register number is used to retrieve the value of that predicate from the predicate register file. This value is then used directly as the branch's prediction.

In any branch prediction scheme, the critical path involves the sequence of recognizing a branch, generating a prediction, and sending that prediction to the fetch hardware. For the POP scheme, the additional lookup of a predicate value may add time to the critical prediction path. In practice, there are many ways to deal with this prediction delay, but all involve tradeoffs between hardware cost and prediction performance. Detailed evaluation of these solutions is beyond the scope of this paper, but two are mentioned here.

One solution is to simply delay the actual prediction for an additional cycle. This would increase the latency of all branches predicted taken by a cycle. However, since more compare instructions could finish execution, this might in fact lead to more accurate branch predictions as more branches may become early-resolved.

Another solution is to reference the BTB with the address preceding each branch at run-time. Since these instructions appear immediately before the branch, it would provide more time for the second predicate lookup. However, in this method there exist address aliasing effects which may degrade prediction accuracy.

With POP, all early-resolved branches will be predicted perfectly. In the case where a predicate is defined late, an earlier value of that predicate will be used. Using this fact, the remaining branches can benefit from a compiler which sets these earlier values meaningfully.

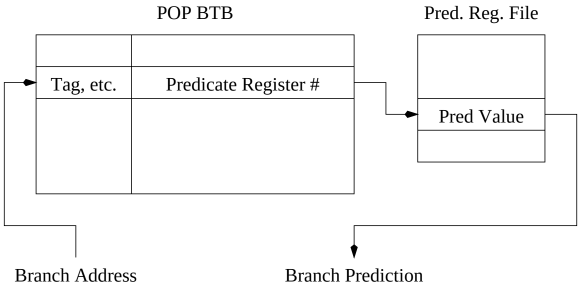


Figure 4: Conceptual design of the Predicate Only Predictor.

3.3 Predicate Enhanced Prediction

POP can perform poorly in a situation where the code has not been compiled for it. This may be a result of either few early-resolved branches or misleading predicate predictors. Another scheme, Predicate Enhanced Prediction (PEP), can deal with this situation gracefully. It can do this because it designed to enhance already aggressive traditional hardware branch predictors only when the predictor predicate is meaningful.

3.3.1 PEP-Counter

The simplest of the PEP schemes is the PEP-Counter. PEP-Counter is an extension of Lee and Smith's two-bit counter scheme [10]. As shown in Figure 5, two two-bit saturating counters are kept for each BTB entry in addition to the predicate register number. When a prediction is needed, the appropriate entry is accessed and the predicate register number is retrieved. Then, the predicate register number is used to access the predicate register file as in the POP scheme. However, the predicate register's value is not used directly as a prediction. Instead, it is used by a multiplexer to pick one of the two saturating counters. One counter, referred to as the *TRUE counter*, is employed when the predicate value is TRUE. The other counter, referred to as the *FALSE counter*, is employed when the predicate is FALSE. As with the Lee and Smith two-bit counter scheme, the chosen counter's highest order bit is used as the prediction. When the actual direction of a predicted branch is known, the counter used for the prediction is updated.

The PEP-Counter scheme is effective in a number of situations. In the early-resolved case, both counters quickly saturate to deliver perfect prediction. This is an important aspect of the PEP-Counter scheme since, as Figure 2 illustrates, a high percentage of branches are early-resolved. The counters are initialized to predict taken for the TRUE counter and not taken for the FALSE counter. This initialization handles all early-resolved branches correctly the first time a prediction is made. The PEP-Counter scheme is also effective when the compiler has not made a prediction or when the compiler makes inaccurate predictions. In such cases, the prediction predicate may seem to be random with respect to the branch direction.

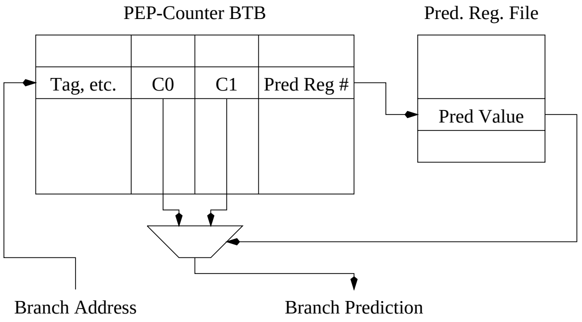


Figure 5: Conceptual design of the Predication Enhanced Prediction Counter scheme.

Here, the two two-bit counters will tend to act as one, making the PEP-Counter act in a manner similar to the two-bit counter scheme.

A case presented later, called partially resolved conditions, is also captured. Here, the branch is always taken when the predicate is TRUE, however, when the predicate is FALSE nothing is known about the branch. In this situation, the TRUE counter will saturate to always correctly predict the branch taken when the predicate is TRUE. When the predicate is FALSE, the PEP-Counter will default to using the FALSE counter in the traditional fashion. All four permutations of partially resolved conditions are handled properly.

3.3.2 PEP-PAs

A more advanced PEP scheme is called PEP-PAs. PEP-PAs is a predication enhanced version of Yeh and Patt's PAs [11]. Instead of one branch history per entry, each PEP-PAs BTB entry contains two histories. The *TRUE history* is utilized when the predictor predicate is TRUE and the *FALSE history* is utilized when the predictor predicate is FALSE. This means that only one of the two histories is used in determining the branch direction and only that same history is updated. By using two histories, the predictor can capture more subtle correlations between branch history, the predictor predicate value, and the actual branch behavior. Figure 6 illustrates the design of a PEP-PAs scheme.

Just like the PEP-Counter, the PEP-PAs's effectiveness stems from the fact that it can capture early-resolved predicates, partially resolved conditions, and good predicate predictions for branches which have them. In addition, when the predicate prediction is not correlated with the branch, the PEP-PAs degenerates to a scheme which behaves similarly to the traditional PAs branch predictor.

4 Opportunities for Predicate-Based Branch Prediction

The branch characteristics of a program can be significantly improved through the use of predicated execution. Architectural support for predicated execution allows the compiler to apply if-conversion which converts

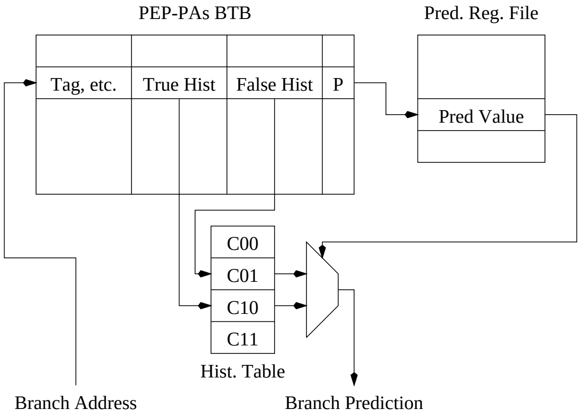


Figure 6: Conceptual design of the Predication Enhanced Prediction PAs scheme.

branch control flow structures into a single block by inserting predicate defining instructions and predicating the original instructions appropriately [12][13][7]. By eliminating branches, if-conversion may lead to a substantial reduction in branch prediction misses and a reduced need to handle multiple branches per cycle for wide issue processors [14][15][16].

Even though predicated execution has been shown to greatly improve the branch characteristics of programs, many situations still exist in which branches remain problematic. In fact, the removal of some branches by if-conversion may adversely affect the predictability of other remaining branches [14]. Consider branch correlation-based predictors. The removal of some branches has the potential to reduce the amount of correlation information available to these predictors. The branches which remain may have little or no correlation among themselves. Another problem is that if-conversion may merge the characteristics of many branches into a single branch, possibly making the new branch harder to predict [14]. The result is that predicated code may need more sophisticated branch prediction.

While predication can make traditional branch predictors less effective, it has the opposite effect on predicate-based predictors. By reducing the static and dynamic branch count, if-conversion increases the distance between branches. This gives the compiler additional freedom to enlarge the distance between branch condition evaluations and their branches. In addition, if-conversion and associated techniques offer natural opportunities for the compiler to generate predictions as described in the rest of this section.

4.1 Restoring Desirable Control Flow Characteristics

Due to resource and hazard constraints, some branches remain after if-conversion [7]. These branches take on a new condition which is the logical-AND of the original condition and the guarding predicate. Giving branches new conditions may have the undesirable effect of making originally easy to predict branches hard to predict.

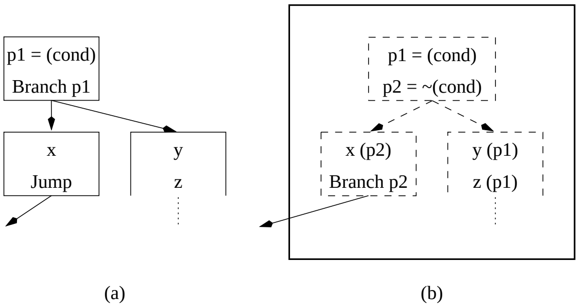


Figure 7: Control flow graph before if-conversion (a) and after if-conversion has merged paths (b).

Figure 7 shows a code segment distilled from a frequently executed segment of the function *elim_lowering* in the benchmark *008.espresso*. The original control flow graph is shown in Figure 7(a). This region is formed into a single predicated block, shown in Figure 7(b), to remove the hard to predict branch in the dominating header block. This branch has a 49.6% fall-through frequency and an unpredictable behavior with respect to traditional predictors. The basic block at the fall through path contains an unconditional branch. During if-conversion two predicates are created. $P1$ is defined to be equal to the original branch condition. $P2$ is the complement of that condition. The destination basic blocks are then predicated upon $p1$ and $p2$. When the unconditional branch is predicated upon $p2$, it effectively becomes a conditional branch with a condition of $p2$. Unfortunately, the hard to predict behavior of the original branch is faithfully transferred to the newly created conditional branch. This branch now has a 49.6% taken frequency and has the same unpredictability as the original branch. For most traditional dynamic branch predictors, the number of total mispredictions is as high as before.

In the newly formed region, the distance between branch condition generation and the branch which uses it is larger than the distance in the original code. If the branch is distanced enough from the predicate defining instruction, the predicate for the new branch will be early-resolved. If this is the case, as in the example, the mispredictions from the original branch can be completely removed through the use of a predicate-based branch predictor. The use of the original branch condition as the predictor restored the originally desirable characteristics of the unconditional branch.

4.2 Previous Predicate Value

Consider a loop with a predicate that corresponds to the back-edge branch condition. Assuming the back-edge branch is not early-resolved in such a loop, the prediction for the back-edge branch will be the branch condition from a previous iteration. If the loop iterates many times, this prediction will be very good as it correctly predicts taken for all but the last iteration. This is one example of the class of opportunities which can benefit from the previous

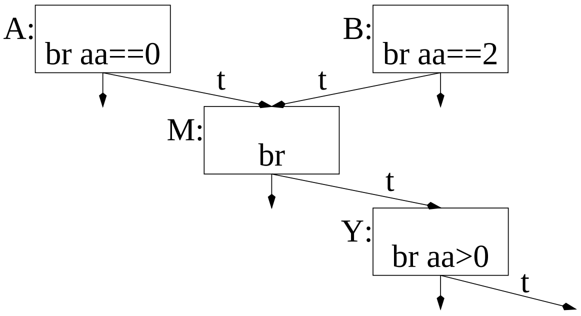


Figure 8: A graph whose paths are indistinguishable by branch history.

predicate value. As with restoring desirable control flow characteristics, the compiler can take advantage of previous predicate values without inserting code to generate the conditions used for predictions.

4.3 Branch Correlation

Any attempt to predict a branch based solely upon that branch's previous behavior is likely at a disadvantage. Branches in a program are often correlated. That is, the history of recently encountered branches may be useful in predicting the one encountered next [17][11]. Many traditional dynamic schemes have been proposed which attempt to capture the correlations among branches. These schemes are successful at achieving a high accuracy of prediction.

However, as Young and Smith pointed out [18], there is still room for additional improvement. The problem with these schemes is that there may be more than one path to a given branch with the same history. In many cases, if the BTB could distinguish paths with the same history, it could make better predictions. Figure 8 shows an example presented by Young and Smith. In this example, there are two paths AMY and BMY which yield the same taken-taken branch history. In path AMY, aa is always 0. This makes the branch in Y always fall through. However, in path BMY, aa is always 2 which makes the branch in Y always taken. With PEP, these paths can be distinguished by setting the predictor predicate in block A to FALSE and to TRUE in block B. By doing this, the PEP schemes make a separate prediction for each path. In fact, in this example setting the predicate to these values will yield perfect prediction for these paths. Young and Smith solved this problem with code duplication. However, the use of predicate-based predictors can reduce the code expansion generated by their method.

As mentioned earlier, the removal of some branches may make the remaining branches harder to predict. Consider the effect of removing, by if-conversion, the branches in blocks A, B, and M from the code in Figure 8. Assuming that these branches delivered useful correlation information before removal, prediction accuracy of the branch in Y would be reduced. Fortunately, using compiler inserted predicate prediction with a PEP scheme, this lost information can be completely restored.

Cycle	Before		After	
0	mul	r2, r1,r1	mul	r2, r1, r1
1	...		pred_gt	p1$_U$,, r1, r4
2	...		...	
3	pred_gt	p1$_U$,, r2, r4	pred_gt	p1$_U$,, r2, r4
4	jump	DEST (p1)	jump	DEST (p1)
		(a)		(b)

Figure 9: Example of architectural correlation.

4.4 Architecture State Correlation

Since all branch directions are determined by program state, it would seem logical that this state would also have sufficient correlation to yield good predictions. Predicate-based schemes could take advantage of such correlation if the information were provided by the compiler. This correlation information can be extracted by profile or expression analysis of the code [8]. While this paper does not attempt to propose methods to find architecture state correlation for branches, it is interesting to study a simple example which hints at their potential.

Figure 9a shows a scheduled segment of code. Assuming the multiply takes three cycles to complete, the actual branch condition cannot be computed until cycle 3. However, a good compiler may be able to place good prediction code earlier. In Figure 9b, a prediction is made in cycle 1. This prediction takes advantage of the fact that $r1$ is an unsigned integer. If $r1$ is greater than $r4$, then $r1^2$ is guaranteed to be greater than $r4$. Here, if the prediction is TRUE, the branch is always taken. Otherwise, PEP will degenerate into a traditional dynamic predictor to cover the $r1$ is not greater than $r4$ case.

4.5 Prediction By Promotion

It is often the case that the condition of a branch is determined by a single expression chain of many predicate defining operations. One way this can occur naturally is by if-converting consecutive branches on a single path. It would be desirable to use the predicate defines which have the most influence on the branch direction for the prediction. Unfortunately, such instructions may be delayed while their predicate source operands are being computed. A solution to the problem of unavailable predicates is known as promotion.

Promotion is a form of speculation which allows an operation to execute before its original predicate is known [7]. It does this by reducing the strength of the predicate so that the operation executes speculatively. If a copy of the predicate defining instruction which has the most influence on the branch condition is promoted, it may be possible to schedule it early enough to use it as a prediction.

Figure 10 is a case in the SPEC benchmark *085.cc1* where the condition with the greatest impact on the branch condition is delayed by its predicate source operand. In this figure, the relative execution weights of the predicated instructions are shown. With unconditional type predicate define instructions, the destination is set to TRUE when the input predicate is TRUE and the condition eval-

1	A	pred_eq	p1$_U$,,r1,0	1.00
2	*D'*	*pred_eq*	*p5$_U$,,r4,100 (p1)*	
2	B	pred_eq	p2$_U$,,r2,100 (p1)	0.69
3	C	pred_eq	p3$_U$,,r3,100 (p2)	0.68
4	D	pred_eq	p4$_U$,,r4,100 (p3)	0.65
5	E	pred_eq	p5$_U$,,r5,100 (p4)	0.30
6	F	jump	DEST (p5)	0.28

Figure 10: Example of prediction by promotion in *085.cc1* with each instruction's probability of execution.

1	A	pred_eq	p1$_U$,,r1,0
2	B	pred_eq	p1$_{OR}$,,r2,0
3	C	pred_eq	p1$_{OR}$,,r3,0
4	D	pred_eq	p1$_{OR}$,,r4,0
5	E	pred_eq	p1$_{OR}$,,r5,0
6	F	jump	DEST (p1)

Figure 11: Example of the branch condition $p1 = (r1 == 0 \mid\mid r2 == 0 \mid\mid r3 == 0 \mid\mid r4 == 0 \mid\mid r5 == 0)$.

uates to TRUE, otherwise, it is set to FALSE. This means that if any condition along this chain evaluates to FALSE, then the remaining predicate definition instructions will all evaluate to FALSE and the branch will not be taken. Execution profiling shows that the percentages of time that predicate defines A, B, C, D, and E evaluate to FALSE are 31%, 1%, 3%, 35%, 2%, respectively. Note that predicate defining instruction D directly controls 35% of the branch occurrences which is significant.

By promoting a copy of operation D as operation D', a good predicate predictor is made. Notice that in cycle 2, where the prediction needs to be made, the value of $p1$ is available. If we predicate D' with $p1$, an even better predicate predictor is made. This is because $p1$ represents the condition of instruction A which is also very restrictive. The combination of the conditions of A and D' yields a perfect prediction of not taken 66% of the time and a highly accurate prediction the other 34%. It is important to note that operation D must remain as it is part of the computation of the correct value of $p5$.

4.6 Partially Resolved Conditions

The condition of a branch is often built from many conditions joined with logical-AND's and logical-OR's. The PlayDoh predicate definition instructions support updating a predicate value by AND'ing or OR'ing to it another condition. The conditions of these predicate defining instructions are usually independent of each other, each bringing the branch predicate closer to the final value used by the branch. At each step, a partially resolved condition exists. By using the partially resolved condition, a prediction can be made. An example of this is shown in Figure 11. The condition of the branch, $(r1 == 0 \mid\mid r2 == 0 \mid\mid r3 == 0 \mid\mid r4 == 0 \mid\mid r5 == 0)$ is generated by OR'ing each subexpression to $p1$.

A perfect prediction can be made for some values of a partially resolved condition. In the example, if any of the predicate definition instructions in the OR-chain writes TRUE, then the computed branch predicate, $p1$ must be TRUE and the branch is taken. However, if no predicate definition instruction has evaluated to TRUE, the branch can only be said to still be based upon the remaining unevaluated conditions. It is important to note, however, that the remaining condition may be easier to predict than the complete condition. For example, $(r4 == 0 \mid\mid r5 == 0)$ may be easier to predict than $(r1 == 0 \mid\mid r2 == 0 \mid\mid r3 == 0 \mid\mid r4 == 0 \mid\mid r5 == 0)$. All predicate-based schemes will capture the case where $p1$ is true early, only the PEP schemes will take advantage of the partial-resolution of the remaining conditions.

A real example of a situation in which partially resolved conditions occur, is contained in the Unix utility *grep*. *Grep* contains a function, *advance*, which is inlined multiple times in various parts of the program. *Advance* consists of a switch/case statement. After loop unrolling, the condition on one of the branches is the logical-OR of many comparisons. While some of the predicate defines occur one cycle before the branch, many others occur up to 15 cycles earlier. Therefore, reading the predicate value for the branch early and using it as a predictor yields significant benefits. In *grep*, 49.4% of all branch mispredictions in the program using traditional predictors reside in only four branches of this type.

4.7 Delaying Branches

Predicate-based predictors predict perfectly if the distance between the branch and the final predicate defining instruction is large enough. In some hard-to-predict branches, a delay of only a few cycles would remove all of the mispredictions by making the branch early-resolved. While the program's cycle count would be penalized by this delay every time the branch is taken, this may be less than the penalty incurred by the branch for all of its mispredictions.

One way to mitigate the loss in performance incurred by delaying a branch is to percolate instructions above the branch. In this way, useful work can continue even though the branch is delayed. In certain cases, where the taken and fall-through paths of the branch are limited by dependence height, a machine with sufficient resources can percolate instructions from both paths without loss of performance to either path. In addition to speculation, predication enables more sophisticated ways of percolating instructions above branches. The Fully Resolved Predication model plus dependence height reduction techniques show great potential for facilitating such percolation [19].

5 Experimental Evaluation

The objective of the experiments presented in this section is to show the potential of the proposed methods. Due to space limitations, these experiments do not address detailed design issues of these schemes.

5.1 Methodology

The target architecture studied in these experiments is a 4-issue in-order superscalar processor that supports

Scheme	Cost Expression	Cost
POP	$n(x + \log_2 p)$	50176
Counter	$n(x + 2)$	46080
PEP-C	$n(x + 4 + \log_2 p)$	54272
PAs	$n(x + h) + 2t(2^h)$	187392
PEP-PAs	$n(x + 2h + \log_2 p) + 2t(2^h)$	205824

Table 1: Hardware cost expressions and net bit cost for schemes evaluated. ($p = 64$, $n = 1024$, $x = 43$, $h = 12$, and $t = 16$).

Var	Definition
p	number of predicate registers
n	number of BTB entries
x	bits in standard BTB entry (target address, tag...)
h	history register size
t	number of history tables

Table 2: Variables and definitions for use in hardware cost computation.

predicated execution and control speculation. No restrictions were placed on the combination of instructions which may be issued together except that only one branch may be issued per cycle and nothing may issue after it. This configuration yields conservative results. Should greater restrictions be placed on instructions which may be issued simultaneously, an extending of the code schedule would result. This longer schedule would improve the results of predicate-based predictors as branches and their conditions would be farther apart. The base architecture is further assumed to have 64 integer, 64 floating-point, and 64 predicate registers. The instruction latencies assumed are those of the HP PA-RISC 7100.

All branch prediction models were based on a 1024-entry direct mapped BTB structure.When simulating the two-level adaptive branch predictors PAs [11] and the PEP-PAs, 12-bit history pattern registers were used and both utilized 16 history tables. All counters were two-bit counters. Table 1 represents a hardware cost estimate of the the five branch prediction models evaluated. This table accounts for bits of hardware memory, but does not include wiring or logic gate costs. Table 2 defines the meaning of symbols found in Table 1. In general, the predicate-based predictors require only slightly more hardware than their traditional branch predictor equivalent.

The effect of two different depths of pipelining on predicate-based predication accuracy was measured. The misprediction penalty is equal to the number of stages before the execution stage (assuming branch prediction is done in the first stage). Verification of a predicated branch involves only the reading of a predicate register and not a logical or arithmetic comparison. Thus, the processor's branch prediction verification is modeled as being performed early in the execution pipeline stage.

High performance code was generated for this target architecture using the IMPACT Compiler version 961030-R. This compiler's support for aggressive speculation, predication, ILP transformations, scheduling, and register allocation was used. Profiling was used to determine important segments of code and to estimate the behavior of branches. The compilation techniques applied are described in detail in [14][20].

It is important to note that the compiler supports a minimal portion of the techniques presented in Section 4. In effect, the baseline performance for these techniques was obtained. The compiler did not insert any prediction code. Instead, an earlier value of the branch's predicate register was used to make the prediction. This captured all early-resolved cases as well as situations where an earlier value of the predicate contained relevant information. It is expected that a compiler that aggressively takes advantage of the enhanced branch prediction hardware would implement many of the techniques presented in Section 4 and generate code with greater branch predication accuracy.

The experimental results presented in this paper were gathered using IMPACT's emulation-driven simulation tools. All the predicated code used in these experiments was emulated on a HP PA-RISC workstation in order to verify proper execution of the predicated code and to drive a detailed cycle by cycle processor simulator. In order to focus on the performance effect of improved branch prediction, perfect instruction and data caches were simulated.

5.2 Results and Analysis

Figures 12 and 13 show the prediction accuracy achieved by various schemes in an architecture where a predicate's value is available to the branch predictor two and four cycles after the predicate definition is fetched. The values in the figures are computed by taking the number of dynamic branches predicted correctly and dividing it by the total number of dynamic branches. The benchmarks studied consist of the six SPEC-92 integer benchmarks and eight Unix utilities: *cccp, cmp, eqn, grep, lex, qsort, wc, and yacc*. However, the benchmarks *wc* and *cmp* were not included in these figures since if-conversion resulted in almost perfect branch prediction.

Figures 14 and 15 show the performance achieved by various schemes. The performance is shown relative to the code executed with perfect branch prediction. Performance is computed by taking the cycle count of the program run with perfect prediction and dividing it by the cycle count of the same program run with each of the schemes presented.

In the predicate-based schemes, prediction accuracy and performance tend to decrease as the pipeline gets deeper. This is due to a decreasing number of early-resolved branches. In addition, there is a reduction in the naturally occurring correlation between the predictor predicate and the branch direction as the predicate is sampled earlier and earlier. The accuracy of the traditional predictors were unaffected by the pipeline length, as expected.

Naturally occurring correlation was helpful in many ways. Previous predicate values were helpful in loops. Restoration of desirable control flow helped remove mispredictions that were introduced during if-conversion. Par-

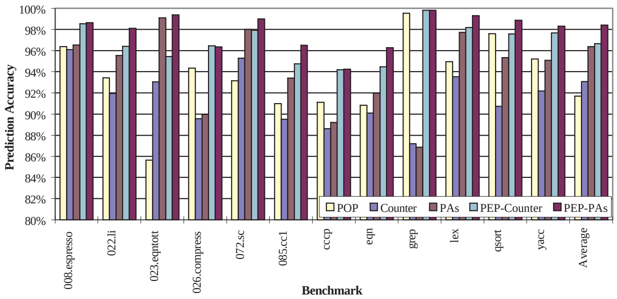


Figure 12: Prediction rate for an architecture where a predicate's value is available to the branch predictor two cycles after the predicate definition is fetched.

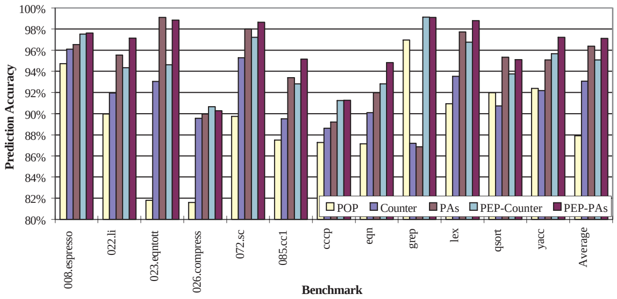


Figure 13: Prediction rate for an architecture where a predicate's value is available to the branch predictor four cycles after the predicate definition is fetched.

tially resolved conditions offered a large reduction in mispredictions to some benchmarks. Unfortunately, these were the only cases discussed in Section 4 that occurred naturally.

There are a few benchmarks which display interesting results. The benchmark *grep* did very well with the predicate-based schemes in comparison to the traditional schemes. This was mostly due to the large amount of partially resolved conditions which existed in *grep*. The reason for this was described in detail in Section 4.6.

Comparison with Figure 2 indicates that much of the accuracy of the predicate-based predictors is not due to early-resolved conditions. An example of a benchmark which derives most of its performance from naturally occurring correlation is *022.li*. At 4 cycles it has only 10% early-resolved branches, yet still has a POP prediction accuracy of 89%. Given this amount of naturally occurring correlation, it appears that there should be a great deal of additional correlation for compilers to extract.

The performance presented here is the baseline performance one would expect from the predicate-based predictors. It is clear that there is potential for performance gains over traditional branch predictors with compiler synthesized predictions. Given that these methods add relatively little additional bit cost in exchange for a modest performance increase now and a potentially large performance increase later, predicate-based predictors should be further studied.

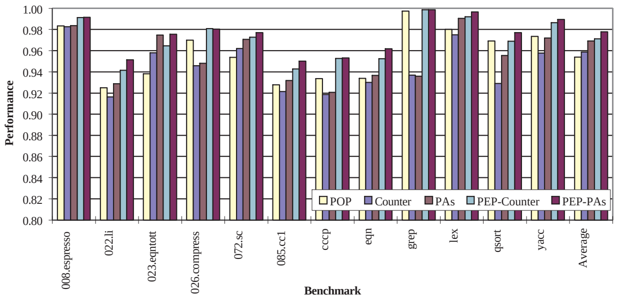


Figure 14: Performance relative to perfect branch prediction for an architecture where a predicate's value is available to the branch predictor two cycles after the predicate definition is fetched. A two cycle mispredict penalty is assumed.

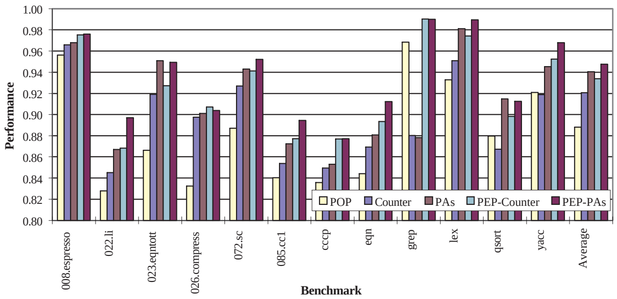


Figure 15: Performance relative to perfect branch prediction for an architecture where a predicate's value is available to the branch predictor four cycles after the predicate definition is fetched. A four cycle mispredict penalty is assumed.

6 Concluding Remarks

The predicate-based branch prediction schemes presented have the potential to significantly enhance the effectiveness of branch prediction. For early-resolved branches, all of the predicate-based predictors offer perfect branch prediction. For other branches, these schemes allow the compiler to generate a prediction or assist traditional branch predictors.

In this paper, we demonstrated that predicated programs tend to have a large portion of branches that are early-resolved. Also, that predicated code contains a significant amount of natural correlation information in the predicates. A compiler providing only minimal support was able to generate a respectable performance improvement.

As we have outlined in Section 4, there are still many great opportunities for future compiler enhancements to further improve overall performance by better utilizing the predicate-based branch prediction schemes.

Acknowledgments

The authors would like to thank all the members of the IMPACT compiler team for their support. Special thanks is due to Scott Mahlke of Hewlett-Packard Laboratories whose numerous insightful comments were very useful. We would also like to thank the anonymous referees for their constructive comments.

References

[1] H. C. Young and J. R. Goodman, "A simulation study of architectural data queues and prepare-to-branch instruction," in *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers ICCD '84*, pp. 544–549, 1984.

[2] R. R. Oehler and R. D. Groves, "IBM RISC System/6000 processor architecture," *IBM Journal of Research and Development*, vol. 34, pp. 23–36, January 1990.

[3] G. R. Beck, D. W. Yen, and T. L. Anderson, "The Cydra 5 minisupercomputer: Architecture and implementation," *The Journal of Supercomputing*, vol. 7, pp. 143–180, January 1993.

[4] V. Kathail, M. S. Schlansker, and B. R. Rau, "HPL PlayDoh architecture specification: Version 1.0," Tech. Rep. HPL-93-80, Hewlett-Packard Laboratories, Palo Alto, CA, February 1994.

[5] D. Weaver, *SPARC-V9 Architecture Specification*. SPARC International Inc., Menlo Park, CA, 1994.

[6] M. Forsyth, S. Mangelsdorf, E. Delano, C. Gleason, and J. Yetter, "CMOS PA-RISC processor for a new family of workstations," in *Proceedings of COMPCON*, pp. 202–207, February 1991.

[7] 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," in *Proceedings of the 25th International Symposium on Microarchitecture*, pp. 45–54, December 1992.

[8] S. A. Mahlke and B. Natarajan, "Compiler synthesized dynamic branch prediction," in *Proceedings of the 29th International Symposium on Microarchitecture*, December 1996.

[9] P. Y. Hsu and E. S. Davidson, "Highly concurrent scalar processing," in *Proceedings of the 13th International Symposium on Computer Architecture*, pp. 386–395, June 1986.

[10] J. E. Smith, "A study of branch prediction strategies," in *Proceedings of the 8th International Symposium on Computer Architecture*, pp. 135–148, May 1981.

[11] T. Y. Yeh and Y. N. Patt, "Two-level adaptive training branch prediction," in *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pp. 51–61, November 1991.

[12] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, pp. 177–189, January 1983.

[13] J. C. Park and M. S. Schlansker, "On predicated execution," Tech. Rep. HPL-91-58, Hewlett Packard Laboratories, Palo Alto, CA, May 1991.

[14] 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," in *Proceedings of the 27th International Symposium on Microarchitecture*, pp. 217–227, December 1994.

[15] D. N. Pnevmatikatos and G. S. Sohi, "Guarded execution and branch prediction in dynamic ILP processors," in *Proceedings of the 21st International Symposium on Computer Architecture*, pp. 120–129, April 1994.

[16] G. S. Tyson, "The effects of predicated execution on branch prediction," in *Proceedings of the 27th International Symposium on Microarchitecture*, pp. 196–206, December 1994.

[17] S. Pan, K. So, and J. T. Rahmeh, "Improving the accuracy of dynamic branch prediction using branch correlation," in *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 76–84, October 1992.

[18] C. Young, N. Gloy, and M. D. Smith, "A comparative analysis of schemes for correlated branch prediction," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 276–286, May 1995.

[19] M. Schlansker and V. Kathail, "Critical path reduction for scalar programs," in *Proceedings of the 28th International Symposium on Microarchitecture*, pp. 57–69, December 1995.

[20] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The Superblock: An effective technique for VLIW and superscalar compilation," *The Journal of Supercomputing*, vol. 7, pp. 229–248, January 1993.