Wish Branches: Combining Conditional Branching and Predication for Adaptive Predicated Execution

Hyesoon Kim § Onur Mutlu § Jared Stark ‡ Yale N. Patt §

§Department of Electrical and Computer Engineering
University of Texas at Austin
{hyesoon,onur,patt}@ece.utexas.edu

‡Oregon Microarchitecture Lab
Intel Corporation
jared.w.stark@intel.com

Abstract

Predicated execution has been used to reduce the number of branch mispredictions by eliminating hard-to-predict branches. However, the additional instruction overhead and additional data dependencies due to predicated execution sometimes offset the performance advantage of having fewer mispredictions.

We propose a mechanism in which the compiler generates code that can be executed either as predicated code or non-predicated code (i.e., code with normal conditional branches). The hardware decides whether the predicated code or the non-predicated code is executed based on a run-time confidence estimation of the branch's prediction. The code generated by the compiler is the same as predicated code, except the predicated conditional branches are NOT removed—they are left intact in the program code. These conditional branches are called wish branches*. The goal of wish branches is to use predicated execution for hard-to-predict dynamic branches and branch prediction for easy-to-predict dynamic branches, thereby obtaining the best of both worlds. We also introduce a class of wish branches, called wish loops, which utilize predication to reduce the misprediction penalty for hard-to-predict backward (loop) branches.*

We describe the semantics, types, and operation of wish branches along with the software and hardware support required to generate and utilize them. Our results show that wish branches decrease the average execution time of a subset of SPEC INT 2000 benchmarks by 14.2% compared to traditional conditional branches and by 13.3% compared to the best-performing predicated code binary.

1. Introduction

Predicated execution has been used to eliminate hard-to-predict branches by converting control dependencies into data dependencies [1]. Traditional predicated execution is not adaptive to run-time (dynamic) branch behavior. The compiler decides to keep a branch as a conditional branch or to predicate it based on compile-time profile information. If the run-time behavior of the branch differs from the compile-time profile behavior, the hardware does not have the ability to override the choice made by the compiler. A predicated branch remains predicated for all its dynamic instances even if it turns out to be very easy-to-predict at run time. Despite the fact that such a branch is rarely mispredicted, the hardware needs to fetch, decode, and execute instructions from *both* control-flow paths. Hence, predicated execution sometimes results in a performance loss because it requires the processing overhead of additional instructions–sometimes without providing any performance benefit.

Figure 1 shows the performance of predicated code on a real system. The results show the execution time of predicated code binaries with different inputs. The data is measured on an Itanium-II machine and binaries are compiled with the ORC-2.0 compiler [22]. Data is normalized to the execution time of a non-predicated code binary for each input. The results show that predicated code binaries generally provide performance benefit over the non-predicated code binaries. But, they sometimes perform worse. For example, for mcf, predicated code provides a 9% performance improvement for input-C, but causes a 4% performance loss for input-A. For bzip2, predicated code only provides a 1% improvement for input-C, but causes a 16% loss for input-A. Hence, the performance of predicated execution is highly dependent on the run-time input set of the program.

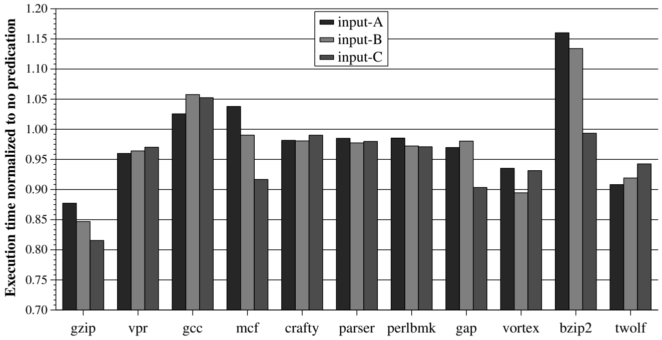


Figure 1. Relative execution time normalized to a non-predicated binary on a real Itanium-II processor.

We would like to eliminate the performance loss due to the overhead of predicated execution by providing a choice to the hardware: the choice of whether or not to use predicated execution for a branch. The compiler is not good at deciding which branches are hard-to-predict because it does not have access to run-time information. In contrast, the hardware has access to accurate run-time information about each branch.

We propose a mechanism in which the compiler generates code that can be executed either as predicated code or non-predicated code (i.e., code with normal conditional branches). The hardware decides whether the predicated code or the non-predicated code is executed based on a run-time confidence estimation of the branch's prediction. The code generated by the compiler is the same as predicated code, except the predicated conditional branches are NOT removed—they are left intact in the program code. These conditional branches are called *wish branches*. When the hardware fetches a wish branch, it estimates whether or not the branch is hard-to-predict using a confidence estimator. If the wish branch is hard-to-predict, the hardware executes the predicated code in order to eliminate a possible branch misprediction. If the wish branch is easy-to-predict, the hardware uses the branch predictor to predict the direction of the wish branch and ignores the predicate information. Hence, wish branches provide the hardware with a way to dynamically choose between conditional branch prediction and predicated execution depending on accurate run-time information about the branch's behavior.

This paper describes the semantics, types, and operation of wish branches. We show that wish branches can eliminate the negative effects of predicated execution and attain better performance than both branch prediction and predicated execution. In Section 2, we provide background information on predicated execution and analyze its overhead. In Section 3, we describe wish branches and their differences from normal branches and predicated execution. Section 4 describes our experimental methodology, and Section 5 presents our experimental results. Section 6 compares our work to related research in the areas of predication, control-flow independence, and multi-path execution.

2. Background and Motivation

Although predicated execution has been implemented in in-order processors [24, 12], as earlier researchers have suggested, the technique can be used in out-of-order processors as well [28, 3, 31, 20, 7, 9]. Since our research aims to reduce the branch misprediction penalty in aggressive high-performance processors, we model predicated execution in an out-of-order processor.[1] This section provides background on the microarchitecture support needed to use predicated execution in an out-of-order processor. We also analyze the overhead of predicated execution in an out-of-order processor to provide motivation for wish branches.

2.1. Microarchitectural Support for Predicated Execution in Out-of-order Execution Processors

In an out-of-order processor, predication complicates register renaming because a predicated instruction may or may not write into its destination register depending on the value of the predicate [28, 23]. Several solutions have been proposed to handle this problem: converting predicated instructions into C-style conditional expressions [28], breaking predicated instructions into two μops [8], the select-μop mechanism [31], and predicate prediction [7]. We briefly describe the baseline mechanism, C-style conditional expressions, we use in this paper. We have also evaluated wish branches on a processor that implements the select-μop mechanism and found similar results, which are described in Section 5.3.3.

Converting a predicated instruction into a C-style conditional expression: In our baseline mechanism, a predicated instruction is transformed into another instruction similar to a C-style conditional expression. For example, `(p1)r1=r2+r3` instruction is converted to the μop `r1=p1?(r2+r3):r1`. If the predicate is TRUE, the instruction performs the computation and stores the result into the destination register. If the predicate is FALSE, the instruction simply moves the old value of the destination register into its destination register, which is architecturally a NOP operation. Hence, regardless of the predicate value, the instruction *always* writes into the destination register, allowing the dependent instructions to be renamed correctly. This mechanism requires four register sources (the old destination register value, the source predicate register, and the two source registers).

2.2. The Overhead of Predicated Execution

Predicated execution reduces the number of branch mispredictions by eliminating hard-to-predict branches. Hard-to-predict branches are eliminated by converting control dependencies into data dependencies using if-conversion [1]. Instructions that are control-dependent on the branch become data-dependent on the branch's predicate (condition) after the branch is eliminated. Because it converts control dependencies into data dependencies, predicated execution introduces two major sources of overhead on the dynamic execution of a program compared to conditional branch prediction. First, the processor needs to fetch additional instructions that are guaranteed to be useless since their predicates will be FALSE. These instructions waste fetch and possibly execution bandwidth and occupy processor resources that can otherwise be utilized by useful instructions. Second, an instruction that is dependent on a predicate value cannot be executed until the predicate value it depends on is ready. This introduces additional delay into the execution of predicated instructions and their dependents, and hence may increase the execution time of the program. We analyze the performance impact of these two sources of overhead on an out-of-order processor model that implements predicated execution. Our simulation methodology and the baseline machine are described in Section 4.

Figure 2 shows the performance improvement achievable if the sources of overhead in predicated execution are ideally eliminated. Data is normalized to the execution time of the non-predicated code binary. For each benchmark, four bars are shown from left to right: (1) BASE-MAX shows the execution time of the predicated code binary produced by the ORC compiler - with all overheads of predicated execution faithfully modeled. (2) NO-DEPEND shows the execution time of the predicated code binary when the dependencies due to predication are ideally (using oracle information) removed. (3) NO-DEPEND + NO-FETCH shows the execution time of the predicated code binary when both sources of overhead in predicated execution are ideally eliminated: in addition to predicate dependencies, the instructions whose predicates are FALSE are ideally eliminated so that they do not consume fetch and execution bandwidth, (4) PERFECT-CBP shows the execution time of the non-predicated code binary when all conditional branches are perfectly predicted using oracle information. This figure shows that predicated execution helps many benchmarks, but it does not improve the *average execution time* over a non-predicated code binary when its overhead is faithfully modeled.[2] However, if the sources of overhead associated with it are completely eliminated, predicated execution would improve the average execution time by 16.4% over no predication. When the overhead of predicated execution is eliminated, the predicated code binary outperforms the non-predicated code binary by more than 2% *on all benchmarks*, even on those where predicated execution normally loses performance (i.e., mcf and bzip2). Note that a significant performance difference still exists between NO-DEPEND + NO-FETCH and PERF-CBP (Perfect conditional branch prediction improves the average execution time by 37.4%). This is due to the fact that not all branches can be eliminated using predication. For example, backward (loop) branches, which constitute a significant proportion of all branches, cannot be eliminated using predicated execution [1, 5].

[1] Note that the mechanism we propose is applicable to any processor that implements predicated execution, regardless of whether it implements in-order or out-of-order scheduling.

[2] The execution time of mcf skews the average normalized execution time, because mcf performs very poorly with predicated execution. Hence, we report two average execution time numbers on our graphs. The set of bars labeled AVG shows the average execution time with mcf included. The set of bars labeled AVGnomcf shows the average execution time with mcf excluded. Since the goal of our mechanism is to reduce the negative effects of predicated execution, we feel that mcf is an important benchmark that needs to be considered in our evaluations.

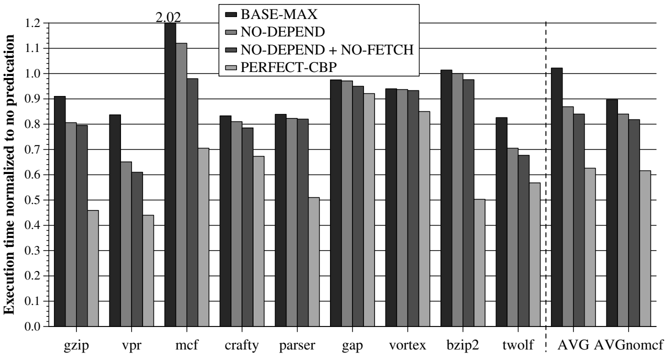
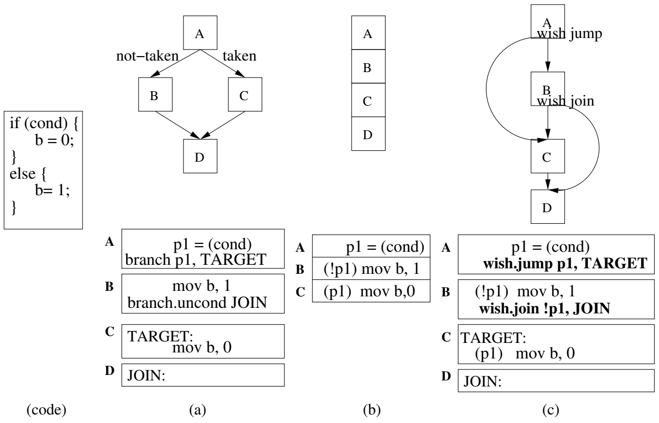


Figure 2. Execution time when sources of overhead in predicated execution are ideally eliminated.



```
if (cond) {
    b = 0;
}
else {
    b= 1;
}
```

```
A   p1 = (cond)
    branch p1, TARGET
B   mov b, 1
    branch.uncond JOIN
C   TARGET:
    mov b, 0
D   JOIN:
```

```
A   p1 = (cond)
B   (!p1) mov b, 1
C   (p1)  mov b,0
```

```
A   p1 = (cond)
    wish.jump p1, TARGET
B   (!p1)  mov b, 1
    wish.join !p1, JOIN
C   TARGET:
    (p1)   mov b, 0
D   JOIN:
```

Figure 3. Source code and the corresponding control flow graphs and assembly code for (a) normal branch code (b) predicated code (c) wish jump/join code.

Our goal in this paper is to design a technique that (1) dynamically reduces the sources of overhead in predicated execution and (2) makes predicated execution applicable to backward branches, thereby increasing the viability and effectiveness of predicated execution in high-performance, out-of-order execution processors. To this end, we propose wish branches. There are three types of wish branches: *wish jumps*, *wish joins*, and *wish loops*. Wish jumps and wish joins are used for forward conditional branches and wish loops are used for backward conditional branches. The goal of wish jumps/joins is to reduce the overhead of predicated execution. The goal of wish loops is to increase the effectiveness of predicated execution by making it applicable to backward branches to reduce the misprediction penalty caused by hard-to-predict loop branches. The next section describes the semantics and operation of wish branches.

3. Wish Branches

3.1. Wish Jumps and Wish Joins

Figure 3 shows a simple source code example and the corresponding control flow graphs and assembly code for: (a) a normal branch, (b) predicated execution, and (c) a wish jump/join. The main difference between the wish jump/join code and the normal branch code is that the instructions in basic blocks B and C are predicated in the wish jump/join code (Figure 3c), but they are not predicated in the normal branch code (Figure 3a). The first conditional branch in the normal branch code is converted to a wish jump instruction and the following control-dependent unconditional branch is converted to a wish join instruction in the wish jump/join code. The difference between the wish jump/join code and the predicated code (Figure 3b) is that the wish jump/join code has branches (i.e., the wish jump and the wish join), but the predicated code does not.

Wish jump/join code can be executed in two different modes (*high-confidence-mode* and *low-confidence-mode*) at run-time. The mode is determined by the confidence of the wish jump prediction. When the processor fetches the wish jump instruction, it generates a prediction for the direction of the wish jump using a branch predictor, just like it does for a normal conditional branch. A hardware confidence estimator provides a confidence estimation for this prediction. If the prediction has high confidence, the processor enters high-confidence-mode. If it has low confidence, the processor enters low-confidence-mode.

High-confidence-mode is the same as using normal conditional branch prediction. To achieve this, the wish jump instruction is predicted using the branch predictor. The source predicate value (p1 in Figure 3c) of the wish jump instruction is predicted based on the predicted branch direction so that the instructions in basic block B or C can be executed before the predicate value is ready. When the wish jump is predicted to be taken, the predicate value is predicted to be TRUE (and block B, which contains the wish join, is not fetched). When the wish jump is predicted to be not taken, the predicate value is predicted to be FALSE and the wish join is predicted to be taken.[3]

Low-confidence-mode is the same as using predicated execution, except it has additional wish branch instructions. In this mode, the wish jump and the following wish join are always predicted to be not taken. The source predicate value of the wish jump instruction is not predicted and the instructions that are dependent on the predicate only execute when the predicate value is ready.[4]

When the confidence estimation for the wish jump is accurate, either the overhead of predicated execution is avoided (high confidence) or a branch misprediction is eliminated (low confidence). When the wish jump is mispredicted in high-confidence-mode, the processor needs to flush the pipeline just like in the case of a normal branch misprediction. However, in low-confidence-mode, the processor never needs to flush the pipeline, even when the branch prediction is incorrect. Like predicated code, the instructions that are not on the correct control flow path will become NOPs since all instructions control-dependent on the branch are predicated.

3.2. Wish Loops

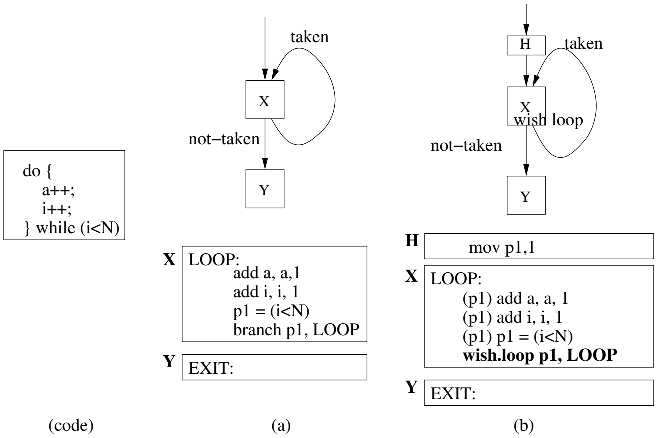
A wish branch can also be used for a backward branch. We call this a *wish loop* instruction. Figure 4 contains the source code for a simple loop body and the corresponding control-flow graphs and assembly code for: (a) a normal backward branch and (b) a wish loop. We compare wish loops only with normal branches since backward branches cannot be directly eliminated using predication [1]. A wish loop uses predication to reduce the branch misprediction penalty of a backward branch without eliminating the branch.

The main difference between the normal branch code (Figure 4a) and the wish loop code (Figure 4b) is that in the wish loop code

[3] Wish join is predicted to be taken because the condition of the wish join is the complement of the wish jump condition in this simple hammock example.

[4] Depending on the microarchitecture design, the predicated instructions could be executed, but the instructions that source the results of the predicated instructions need to wait until the predicate value is ready.

the instructions in block X (i.e., the loop body) are predicated with the loop branch condition. Wish loop code also contains an extra instruction in the loop header to initialize the predicate to 1 (TRUE). To simplify the explanation of the wish loops, we use a `do-while` loop example in Figure 4. A `while` loop can also utilize a wish loop instruction as shown in Figure 5. Similarly, a `for` loop can also utilize a wish loop instruction.



```
do {
    a++;
    i++;
} while (i<N)
```

```
X  LOOP:
          add a, a,1
          add i, i, 1
          p1 = (i<N)
          branch p1, LOOP
Y  EXIT:
```

```
H         mov p1,1
X  LOOP:
        (p1) add a, a, 1
        (p1) add i, i, 1
        (p1) p1 = (i<N)
        wish.loop p1, LOOP
Y  EXIT:
```

Figure 4. `do-while` **loop source code and the corresponding control flow graphs and assembly code for (a) normal backward branch code (b) wish loop code.**

```
while (i<N) {
    a++;
    i++;
}
```
(code)

```
LOOP:   p1 = (i<N)
        branch !p1, EXIT

        add a, a,1
        add i, i, 1
        branch.uncond LOOP
EXIT:
```
(a)

```
             p1 = (i<N)

LOOP: (p1) add a, a,1
      (p1) add i, i, 1
      (p1) p1 = (i<N)
           wish.loop p1, LOOP
```
(b)

Figure 5. `while` **loop source code and assembly code for (a) normal backward branch code (b) wish loop code.**

When the wish loop instruction is first encountered, the processor enters either high-confidence-mode or low-confidence-mode, depending on the confidence of the wish loop prediction.

In high-confidence-mode, the processor predicts the direction of the wish loop according to the loop/branch predictor. If the wish loop is predicted to be taken, the predicate value (p1 in Figure 4b) is predicted to be TRUE, so the instructions in the loop body can be executed without waiting for the predicate to become ready. If the wish loop is mispredicted in high-confidence-mode, the processor flushes the pipeline, just like in the case of a normal branch misprediction.

If the processor enters low-confidence-mode, it stays in this mode until the loop is exited. In low-confidence-mode, the processor still predicts the wish loop according to the loop/branch predictor. However, it does *not* predict the predicate value. Hence, the iterations of the loop are predicated (i.e., fetched but not executed until the predicate value is known) during low-confidence-mode. There are three misprediction cases in this mode: (1) *early-exit*: the loop is iterated fewer times than it should be, (2) *late-exit*: the loop is iterated only a few more times by the processor front end than it should be and the front end has already exited when the wish loop misprediction is signalled, and (3) *no-exit*: the loop is still being iterated by the processor front end when the wish loop misprediction is signalled (as in the late-exit case, it is iterated more times than needed).

For example, consider a loop that iterates 3 times. The correct loop branch directions are TTN (taken, taken, not-taken) for the three iterations, and the front end needs to fetch blocks $X_1X_2X_3Y$, where X_i is the i^{th} iteration of the loop body. An example for each of the three misprediction cases is as follows: In the early-exit case, the predictions for the loop branch are TN, so the processor front end fetches blocks X_1X_2Y. One example of the late-exit case is when the predictions for the loop branch are TTTTN so the front end fetches blocks $X_1X_2X_3X_4X_5Y$. For the no-exit case, the predictions for the loop branch are TTTTT...T so the front end fetches blocks $X_1X_2X_3X_4X_5...X_N$.

In the early-exit case, the processor needs to execute X at least one more time (in the example above, exactly one more time; i.e., block X_3), so it flushes the pipeline just like in the case of a normal mispredicted branch.

In the late-exit case, the fall-through block Y has been fetched before the predicate for the first extra block X_4 has been resolved. Therefore it is more efficient to simply allow X_4 and subsequent extra block X_5 to flow through the data path as NOPs (with predicate value p1 = FALSE) than to flush the pipeline. In this case, the wish loop performs better than a normal backward branch because it reduces the branch misprediction penalty. The smaller the number of extra loop iterations fetched, the larger the reduction in the branch misprediction penalty.

In the no-exit case, the front end has not fetched block Y at the time the predicate for the first extra block X_4 has been resolved. Therefore, it makes more sense to flush X_4 and any subsequent fetched extra blocks, and then fetch block Y, similar to the action taken for a normal mispredicted branch. We could let $X_4X_5...X_N$ become NOPs as in the late-exit case, but that would increase energy consumption without improving performance.

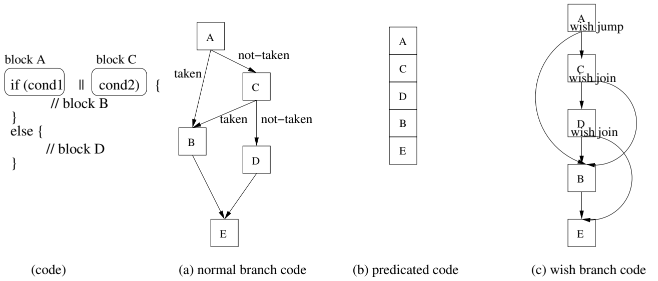
We expect wish loops to do well in integer benchmarks where loops iterate a small but variable number of times in an unpredictable manner [10]; e.g., loops that cannot be captured by a loop branch predictor [27]. As wish loops reduce the misprediction penalty for the late-exit case, a specialized wish loop predictor can be designed to predict wish loop instructions. This predictor does not have to exactly predict the iteration count of a loop. It can be biased to overestimate the iteration count of a loop to make the late-exit case more common than the early-exit case for a hard-to-predict wish loop.

3.3. Wish Branches in Complex Control Flow

Wish branches are not only used for simple control flow. They can also be used in complex control flow where there are multiple branches, some of which are control-dependent on others. Figure 6 shows a code example with complex control flow and the control flow graphs of the normal branch code, predicated code, and the wish branch code corresponding to it.

When there are multiple wish branches in a given region[5], the first wish branch is a wish jump and the following wish branches are wish joins. We define a wish join instruction to be a wish branch instruction that is control-flow dependent on another wish branch instruction. Hence, the prediction for a wish join is dependent on the confidence estimations made for the previous wish jump, any previous wish joins, and the current wish join itself. If the previous wish jump, any of the previous wish joins, or the current wish join is low-confidence, the current wish join is predicted to be not-taken. Oth-

[5] A region is a single basic block or a set of basic blocks [17].



```
block A          block C
if (cond1   ||   cond2)  {
         // block B
}
else {
        // block D
}
```

Figure 6. Control flow graph examples with wish branches.

erwise, the current wish join is predicted using the branch predictor. An example of the predictions made for each of the wish branches in Figure 6c is shown in Table 1.[6]

Table 1. The prediction of multiple wish branches in Figure 6c.

confidence			prediction		
jump (A)	join (C)	join (D)	jump (A)	join (C)	join (D)
high	high	high	predictor	predictor	predictor
high	high	low	predictor	predictor	not-taken
high	low	-	predictor	not-taken	not-taken
low	-	-	not-taken	not-taken	not-taken

3.4. ISA Support for Wish Branches

We assume that the baseline ISA to which wish branches are to be added supports predicated execution. If the current ISA already has unused hint bits for the conditional branch instruction, like the IA-64 [12], wish branches can be implemented using the hint bit fields without modifying the ISA. Figure 7 shows a possible instruction format for the wish branch. A wish branch can use the same opcode as a normal conditional branch, but its encoding has two additional fields: *btype* and *wtype*. If the processor does not implement the hardware support required for wish branches, it can simply treat a wish branch as a normal branch (i.e., ignore the hint bits). New binaries containing wish branches will run correctly on existing processors without wish branch support.

OPCODE	btype	wtype	target offset	p

btype: branch type (0:normal branch 1:wish branch)
wtype: wish branch type (0:jump 1:loop 2:join)
p: predicate register identifier

Figure 7. A possible instruction format for the wish branch.

3.5. Hardware Support for Wish Branches

Aside from the hardware to support predicated execution, wish branches require the hardware support described below.

3.5.1. Instruction fetch and decode hardware Instruction decode logic needs to be modified so that wish branch instructions can be decoded. A BTB entry is extended to indicate whether or not the branch is a wish branch and the type of the wish branch. The fetch logic requires one additional mux to override the result of the branch predictor for a wish jump or a wish join in low-confidence-mode (since a wish jump or join is always predicted not-taken in low-confidence-mode regardless of the branch predictor outcome).

3.5.2. Wish branch state machine hardware Figure 8 shows the front-end state machine that manages the various modes of a processor implementing wish branches. There are three modes: normal-mode `(00)`, low-confidence-mode `(10)`, and high-confidence-mode `(01)`. The state diagram summarizes the mode transitions that occur in the front-end of a processor supporting wish branches, based on the information provided in Sections 3.1 and 3.2.

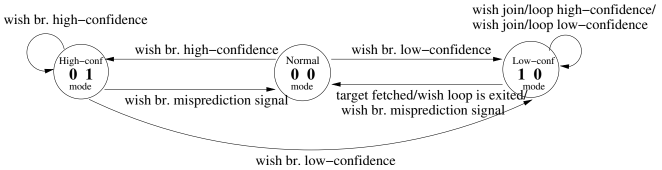

wish br. high−confidence
wish join/loop high−confidence/
wish join/loop low−confidence
wish br. high−confidence
wish br. low−confidence
High−conf
0 1
mode
Normal
0 0
mode
Low−conf
1 0
mode
wish br. misprediction signal
target fetched/wish loop is exited/
wish br. misprediction signal
wish br. low−confidence

Figure 8. State diagram showing mode transitions in a processor that supports wish branches. "target fetched" means that the target of the wish jump/join that caused entry into low-confidence-mode is fetched.

3.5.3. Predicate dependency elimination module As we described in Sections 3.1 and 3.2, the predicate of the wish branch is predicted during high-confidence-mode to eliminate the delay in the execution of predicated instructions. To support this, when the processor enters high-confidence-mode, the predicate register number of the wish branch instruction is stored in a special buffer. Each following instruction compares its source predicate register number with the register number in the special buffer. If both predicate register numbers are the same, the source predicate register of the instruction is assumed to be ready, with a TRUE value when the wish branch is predicted to be taken and with a FALSE value when the wish branch is predicted to be not taken. The special buffer is reset if there is a branch misprediction or if an instruction that writes to the same predicate register is decoded.

3.5.4. Branch misprediction detection/recovery module When a wish branch misprediction is detected, the processor needs to decide whether or not a pipeline flush is necessary. If the wish branch is mispredicted during high-confidence-mode,[7] the processor always flushes the pipeline. If the wish branch is mispredicted during low-confidence-mode and the wish branch is a wish jump or a wish join, then the processor does not flush the pipeline.

If a wish loop is mispredicted during low-confidence-mode, the processor needs to distinguish between early-exit, late-exit, and no-exit. To support this, the processor uses a small buffer in the front end that stores the last prediction made for each static wish loop instruction that is fetched but not yet retired. When a wish loop is predicted, the predicted direction is stored into the entry corresponding to the static wish loop instruction. When a wish loop is found to be mispredicted and the actual direction is taken, then it is an early-exit case. So, the processor flushes the pipeline. When a wish loop is mispredicted and the actual direction is not-taken, the branch misprediction recovery module checks the latest prediction made for the same static wish loop instruction by reading the buffer in the front end. If the last stored prediction is not taken, it is a late-exit case, because the front end must have already exited the loop, so no pipeline flush is required. If the last stored prediction is taken, it is a no-exit case because the front-end must still be fetching the loop body, and

[6] Note that the high-high-low case should not happen with a good confidence estimator in this given example. Since the branch condition of join C is the complement of the condition of join D, a good confidence estimator will estimate join D to be high confidence if join C is estimated high confidence.

[7] The mode that is checked when a wish branch is mispredicted is the mode of the front-end when that branch was fetched, *not* the mode of the front-end at the time the misprediction is detected.

the processor flushes the pipeline.[8] To keep the hardware simple we do not support nested wish loops.

3.5.5. Confidence estimator An accurate confidence estimator is essential to maximize the benefits of wish branches. An inaccurate confidence estimation for a wish branch can be harmful in two different ways. First, if the wish branch prediction is estimated to be low confidence even though the prediction is correct, the processor suffers from the overhead of predicated execution without any performance benefit. Second, if the wish branch prediction is estimated to be high confidence when the branch is actually mispredicted, the processor loses the opportunity to eliminate a pipeline flush.

Previously proposed confidence estimators, such as the JRS confidence estimator [13], can be used to estimate the confidence of wish branch predictions. In our evaluations, we used a modified JRS estimator. Since the confidence estimator is dedicated to wish branches, its size is small. If the baseline processor already employs a confidence estimator for normal conditional branches, this estimator can also be utilized to estimate the confidence of wish branch predictions.

3.6. Compiler Support for Wish Branch Generation

A wish branch binary is an object file consisting of a mixture of wish branches, traditional predicated code, and normal branches. The compiler decides which branches are predicated, which are converted to wish branches, and which stay as normal branches based on estimated branch misprediction rates and compile-time heuristics. The compile-time decisions need to take into account the following:

1. The size and the execution time of the basic blocks that are considered for predication/wish branch code.
2. Input data set dependence/independence of the branch.
3. The estimated branch misprediction penalty.
4. The extra instruction overhead associated with predicated execution or wish branches.

For example, it may be better to convert a short forward branch which has only one or two control-dependent instructions into predicated code rather than wish branch code because wish branch code has the overhead of at least one extra instruction (i.e., the wish jump instruction). If the misprediction rate of a branch is strongly dependent on the input data set, the compiler is more apt to convert the code into wish branch code. Otherwise, the compiler is more apt to use a normal branch or convert the code into predicated code. The compiler can determine whether or not the misprediction rate is dependent on the input data with heuristics. The compiler heuristics used to decide which branches should be converted into wish branches is an important research area that we intend to investigate in future work. The heuristics we used for our initial performance evaluations are described in Section 4.2.

We note that wish branches provide the compiler with more flexibility in generating predicated code. With wish branches, if the compiler makes a "bad decision" at compile time, the hardware has the ability to "correct" that decision at run time. Hence, the compiler can generate predicated code more aggressively and the heuristics used to generate predicated code can be less complicated.

3.7. Advantages/Disadvantages of Wish Branches

In summary, the advantages of wish branches are as follows:

1. Wish branches provide a way to reduce the negative effects of predicated code. The performance of the wish branch code could be the best of normal branch (non-predicated) code and predicated code, regardless of variations in the input set of a program.
2. Wish branches increase the benefits of predicated code, because they allow the compiler to generate more aggressively predicated code. For example, the compiler can generate predicated code (with wish branches) for larger blocks because the overhead of predication can be avoided dynamically if the wish branch turns out to be easy-to-predict.
3. Unlike traditional predicated execution, wish branches provide a mechanism to exploit predication to reduce the branch misprediction penalty for *backward* branches.

The disadvantages of wish branches compared to predication are:

1. Wish branches require extra branch instructions. These instructions would take up machine resources and instruction cache space. However, the larger the predicated code block, the less significant this becomes.
2. The extra wish branch instructions increase the contention for branch predictor table entries. This may increase negative interference in the pattern history tables. We found that performance loss due to this effect is negligible.
3. Wish branches reduce the size of the basic blocks by adding control dependencies into the code. Larger basic blocks can provide better opportunity for compiler optimizations, such as instruction scheduling. If the compiler used to generate wish branch binaries is unable to perform aggressive code optimizations across basic blocks, the presence of wish branches may constrain the compiler's scope for code optimization.

4. Methodology

Figure 9 illustrates our simulation infrastructure. We chose the IA-64 ISA to evaluate the wish branch mechanism, because of its full support for predication, but we converted the IA-64 instructions to micro-operations (μops) to execute on our out-of-order superscalar processor model. We modified the ORC compiler [22] to generate the IA-64 binaries (with and without wish branches). The binaries were then run on an Itanium II machine using the Pin binary instrumentation tool [18] to generate traces. These IA-64 traces were later converted to μops. The μops were fed into a cycle-accurate simulator to obtain performance results.

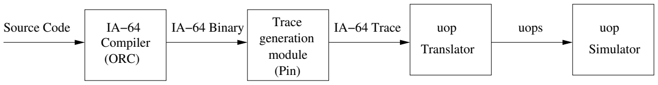


Figure 9. Simulation infrastructure

4.1. μop Translator and Simulator

We developed an IA-64 translator which converts the disassembled IA-64 instructions into our simulator's native μops. We model μops to be close to a generic RISC ISA. Our translator handles correctly all the issues related to IA-64 specific features such as rotating registers. All NOPs are eliminated during μop translation.

[8] If the processor exited the loop and then re-entered it, this case will be incorrectly identified as a no-exit case, when it is actually a late-exit case. Hence, the processor unnecessarily flushes the pipeline, but it still functions correctly. We did not see this case happen in the benchmarks we simulated.

μops are fed into our cycle-accurate simulator. Our baseline processor is an aggressive superscalar, out-of-order processor. Table 2 describes our baseline micro-architecture. Because a less accurate branch predictor would provide more opportunity for wish branches, a very large and accurate hybrid branch predictor is used in our experiments to avoid inflating the impact of wish branches.

Table 2. Baseline processor configuration

Front End	64KB, 4-way, 2-cycle I-cache; 8-wide fetch/decode/rename Fetches up to 3 cond. br. but fetch ends at the first taken br. I-cache stores IA-64 instructions; decoder/ROM produces μops
Branch Predictors	64K-entry gshare [21]/PAs [32] hybrid, 64K-entry selector 4K-entry BTB; 64-entry RAS; 64K-entry indirect target cache minimum branch misprediction penalty is 30 cycles
Execution Core	512-entry reorder buffer; 8-wide execute/retire
On-chip Caches	L1 data cache: 64KB, 4-way, and 2-cycle latency L2 unified cache: 1MB, 8-way, 8 banks, 6-cycle latency All caches use LRU replacement and have 64B line size
Buses and Memory	300-cycle minimum memory latency; 32 memory banks 32B-wide core-to-memory bus at 4:1 frequency ratio
Predication support	Converted into C-style conditional expressions [28]
Confidence estimator	1KB, tagged (4-way), 16-bit history JRS estimator [13]

4.2. Compilation

All benchmarks were compiled for the IA-64 ISA with the -O2 optimization by the ORC compiler. Software pipelining, speculative loads, and other IA-64 specific optimizations were turned off to reduce the effects of features that are specific to the IA-64 ISA and that are less relevant on an out-of-order microarchitecture. Software pipelining was shown to provide less than 1% performance benefit on the SPEC CPU2000 INT benchmarks [5] and we removed this optimization to simplify our analysis. Wish branch code generation is also performed with -O2 optimization. To compare wish branches to normal branches and predication, we generated five different binaries for each benchmark, which are described in Table 3. Unless otherwise noted, all execution time results reported in this paper are normalized to the execution time of the normal branch binaries. Section 4.2.1 and 4.2.2 briefly describe the compilation algorithms we use in our experiments.

4.2.1. Predicated code binary generation algorithm To generate predicated code, the ORC compiler first checks whether or not the control-flow graph is suitable for if-conversion in a region boundary. The ORC compiler performs if-conversion within a region boundary. When the control-flow graph is suitable for if-conversion, the compiler calculates the following equations. Each probability in these equations is determined using compiler heuristics. Execution times are estimated with dependency height and resource usage analysis. We set the branch misprediction penalty to 30 cycles. In the BASE-DEF binary, branches which satisfy Equation (4.3) are converted to predicated code. In the BASE-MAX binary, all branches that are suitable for if-conversion are converted to predicated code. Hence, the BASE-MAX binary contains more aggressively predicated code. We use two predicated code binaries as our baselines because neither binary performs the best for all benchmarks: for some benchmarks BASE-DEF performs better and for others BASE-MAX performs better.

$Execution\ time\ of\ normal\ branch\ code = exec\_T * P(T) + exec\_N * P(N) + misp\_penalty * P(misprediction)$ (Equation 4.1)

$Execution\ time\ of\ predicated\ code = exec\_pred$ (Equation 4.2)

$Exec.\ time\ of\ predicated\ code < Exec.\ time\ of\ normal\ br.\ code$ (Equation 4.3)

$exec\_T$: Exec. time of the code when the br. under consideration is taken
$exec\_N$: Exec. time of the code when the br. under consideration is not taken
$P(case)$: The probability of the case; e.g., P(T) is the prob. that the br. is taken
$misp\_penalty$: Machine-specific branch misprediction penalty
$exec\_pred$: Execution time of the predicated code

4.2.2. Wish branch binary generation algorithm If a branch is suitable for if-conversion, we treat that branch as a wish branch candidate. If the number of instructions in the fall-through block of a branch is greater than N (we set N to 5), the candidate branch is converted to a wish jump and the necessary wish joins are inserted. Otherwise, the wish branch candidate is converted to predicated code. We use a threshold of 5 instructions because we estimate that very short forward branches are better off being predicated. A loop branch is converted into a wish loop if the number of instructions in the loop body is less than L (we set L to 30). We have not tuned the thresholds N and L used in these heuristics. Since our baseline compiler is not optimized to build large predicated code blocks, we inserted some of the wish branches using a binary instrumentation tool when the control flow is suitable to be converted to wish branch code.

4.3. Trace Generation and Benchmarks

IA-64 traces were generated with the Pin instrumentation tool [18]. Because modeling wrong-path instructions is important in studying the performance impact of wish branches, we generated traces that contain wrong-path information by forking a wrong-path trace generation thread. We forked a thread at every wish branch down the mispredicted path. The spawned thread executed until the number of executed wrong-path instructions exceeded the instruction window size. The trace contains the PC, predicate register, register value, memory address, binary encoding, and the current frame marker information for each instruction.

All experiments were performed using the SPEC INT 2000 benchmarks. The benchmarks were run with a reduced input set [16] to simulate until the end of the program. Table 4 shows information about the simulated benchmarks for the normal branch binaries and the wish jump/join/loop binaries.[9] Branch information displayed is collected only for conditional branches. For the wish jump/join/loop binaries, we show the total number of static and dynamic wish branches and the percentage of wish loops among all wish branches.

5. Simulation Results and Analysis

5.1. Wish Jumps/Joins

We first evaluate how using wish jumps/joins performs compared to normal branches and predicated code. Figure 10 shows the normalized execution time of four different configurations for each benchmark: (1) BASE-DEF binary, (2) BASE-MAX binary, (3) wish jump/join binary with a real JRS confidence estimator, and (4) wish jump/join binary with a perfect confidence estimator. With a real confidence estimator, the wish jump/join binaries improve the average execution time by 11.5% over the normal branch binaries and by 10.7% over the best-performing (on average) predicated code binaries (BASE-DEF). The wish jump/join binaries perform better than the normal branch binaries for all the benchmarks, except mcf. Moreover, they perform better than both of the predicated code binaries for gzip, vpr, mcf, gap, and, twolf. For vpr, mcf, and twolf - three benchmarks where the overhead of predicated execution is very high,

[9] Due to problems encountered during trace generation using Pin, gcc, perlbmk and eon benchmarks were excluded. NOPs *are* included in the dynamic IA-64 instruction count, but they are *not* included in the μop count.

Table 3. Description of binaries compiled to evaluate the performance of different combinations of wish branches

Binary name	Branches that can be predicated with the ORC algorithm [17, 22, 20] ...	Backward branches ...
normal branch binary	remain as normal branches	remain as normal branches
predicated code binary: BASE-DEF	are predicated based on the compile-time cost-benefit analysis	remain as normal branches
predicated code binary: BASE-MAX	are predicated	remain as normal branches
wish jump/join binary	are converted to wish jumps/joins or are predicated	remain as normal branches
wish jump/join/loop binary	are converted to wish jumps/joins or are predicated	are converted to wish loops or remain as normal branches

Table 4. Simulated benchmarks

	Normal branch binary					Wish jump/join/loop binary	
Benchmark	Dynamic instructions IA64 instructions / μops	Static branches	Dynamic branches	Mispredicted branches (per 1000 μops)	IPC/μPC	Static wish branches (% of wish loops)	Dynamic wish branches (% of wish loops)
164.gzip	303M / 211M	1271	31M	8.3	2.25/ 1.53	93 (80%)	9.5M (61%)
175.vpr	161M / 106M	4078	13M	7.8	2.38/ 1.60	206 (83%)	4.3M (35%)
181.mcf	189M / 135M	1288	28M	4.7	1.52/ 1.46	31 (54%)	5.1M (20%)
186.crafty	316M / 227M	4334	30M	4.7	1.68/ 1.01	271 (65%)	3.7M (49%)
197.parser	428M / 311M	2879	72M	9.6	1.21/ 0.87	214 (88%)	14.2M (63%)
254.gap	611M / 423M	4163	50M	1.0	1.22/ 0.80	167 (74%)	6.1M (75%)
255.vortex	113M / 87M	7803	12M	0.8	1.06/ 0.84	104 (33%)	1.7M (62%)
256.bzip2	429M / 308M	1236	40M	8.6	1.38/ 1.37	130 (81%)	8.7M (90%)
300.twolf	171M / 114M	4306	10M	6.8	1.81/ 1.16	356 (71%)	3.1M (57%)

as was shown in Figure 2 - the wish jump/join binaries improve the execution time by more than 10% over the predicated code binaries.

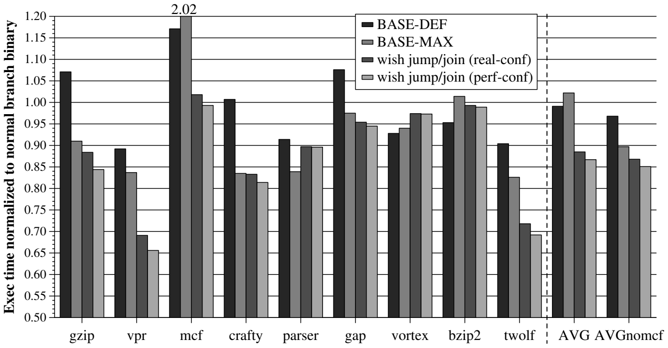


Figure 10. Performance of wish jump/join binaries

Figure 10 also shows that the wish jump/join binaries reduce the overhead which causes the predicated code binaries to perform worse than the normal branch binaries. For example, the BASE-DEF binaries perform worse than the normal branch binaries for gzip, mcf, crafty, and gap. Similarly, the BASE-MAX binaries perform worse than the normal branch binaries on mcf and bzip2. In fact, aggressive predication (BASE-MAX) increases the execution time of mcf by 102% because of the additional delay caused by predicated instructions. In mcf, the execution of many critical load instructions that would cause cache misses are delayed because their source predicates are dependent on other critical loads which incur cache misses. Hence, predicated execution results in the serialization of many critical load instructions that would otherwise be serviced in parallel had branch prediction been used, leading to a large performance degradation. The wish jump/join binaries eliminate the performance loss due to predicated execution on benchmarks where predicated execution reduces performance. Hence, wish branches are effective at reducing the negative effects of predicated execution.

The wish jump/join binary performs worse than both of the predicated code binaries only for one benchmark, vortex. This is due to the reduced size of the basic blocks in the wish jump/join binary for vortex. The compiler is able to optimize the code better and more aggressively in the predicated code binaries that have larger basic blocks. Note that the compiler heuristics we used to insert wish branches are very simple. Better heuristics that take into account more information, as explained in Section 3.6, can eliminate the disadvantages caused by wish branches in vortex.

Figure 11 shows the dynamic number of wish branches per 1 million retired μops. The left bar for each benchmark shows the number of wish branches predicted to have low-confidence and how many of those were mispredicted. The right bar shows the number of wish branches predicted to have high-confidence and how many of those were mispredicted. Ideally, we would like two conditions to be true. First, only the actually mispredicted wish branches should be estimated as low-confidence. Second, no mispredicted wish branch should be estimated as high-confidence. Figure 11 shows that the second condition is much closer to being satisfied than the first on all benchmarks. Very few of the high-confidence branches are actually mispredicted. However, the first condition is far from being satisfied, especially in gzip, vpr, mcf, crafty, and twolf. In these benchmarks, a significant number of wish branches are estimated as low-confidence even though they are not mispredicted. Therefore, a better confidence estimator would improve the performance of wish branches on these benchmarks, as shown in the rightmost bars in Figure 10.

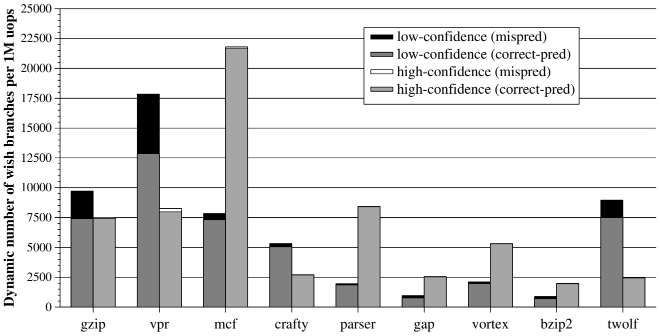


Figure 11. Dynamic number of wish branches per 1M retired μops. Left bars: low-confidence, right bars: high-confidence.

Figure 11 also provides insight into why wish branches improve the performance of predicated execution significantly in some benchmarks. For example, in mcf most of the branches that are converted to wish branches are correctly predicted. These branches are predicated in the BASE-MAX binary. However, predicating them reduces the performance with the reduced input set, because those branches are almost always correctly predicted. Converting them into wish branches rather than predicating them allows the hardware to dynamically decide whether or not they should be predicated. As shown in Figure 11, the hardware confidence estimator does well on mcf and correctly identifies most of the correctly-predicted wish branches as high-confidence. Hence, for those wish branches, the overhead of predicated execution is avoided and the wish branch binary performs as well as the normal branch binary. Similarly in gzip, vpr, and gap, many of the wish branches are correctly predicted and also estimated as high confidence, resulting in significant savings in the overhead of predicated execution, which is reflected in the performance of the wish jump/join binaries for these three benchmarks in Figure 10. Most wish branches are correctly predicted and identified as high-confidence also in parser and vortex. However, the performance of parser and vortex is not improved with wish branches compared to the predicated code binaries, because the overhead of predicated execution is very low for these two benchmarks as shown in Figure 2.

5.2. Wish Jumps/Joins and Wish Loops

Figure 12 shows the performance of wish branches when wish loops are also used in addition to wish jumps/joins. With a real confidence estimator, the wish jump/join/loop binaries improve the average execution time by 14.2% compared to the normal branch binaries and by 13.3% compared to the best-performing (on average) predicated code binaries (BASE-DEF). An improved confidence estimator has the potential to increase the performance improvement up to 16.2% compared to the normal branch binaries. Even if mcf is excluded from the calculation of the average execution time, the wish jump/join/loop binaries improve the average execution time by 16.1% compared to the normal branch binaries and by 6.4% compared to the best-performing predicated binaries (BASE-MAX), with a real confidence estimator.

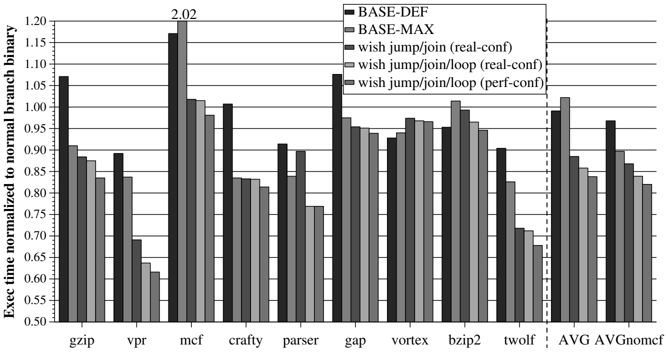


Figure 12. Performance of wish jump/join/loop binaries

Using wish loops in addition to wish jumps/joins improves the execution time of vpr, parser, and bzip2 by more than 3%. The reason for the performance improvement on these three benchmarks can be seen in Figure 13. This figure shows the dynamic number of wish loops per 1 million μops and classifies them based on their confidence estimation and misprediction status. Remember that the *late-exit* misprediction case is the only case where a wish loop improves performance compared to a normal loop branch, as described in Section 3.2. In vpr, parser, and bzip2 there is a significant number of wish loop instructions that are predicted to be low-confidence and are actually mispredicted as *late-exit*. Therefore, we see significant performance improvements due to wish loops for these benchmarks.

We also compare the performance of wish branches to the best-performing binary for each benchmark. To do so, we selected the best-performing binary for each benchmark among the normal branch binary, BASE-DEF predicated code binary, and BASE-MAX predicated code binary based on the execution times of these three binaries, which are obtained via simulation. Note that this comparison is unrealistic, because it assumes that the compiler can, at compile-time, predict which binary would perform the best for the benchmark at run-time. This assumption is not correct, because the compiler does not *know* the run-time behavior of the branches in the program. Even worse, the run-time behavior of the program can also vary from one run to another. Hence, depending on the input set to the program, a different binary could be the best-performing binary, as we have already shown in Figure 1.

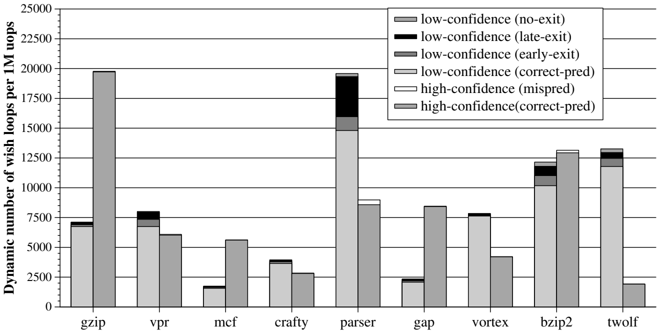


Figure 13. Dynamic number of wish loops per 1M retired μops. Left bars: low-confidence, right bars: high-confidence.

Table 5 shows, for each benchmark, the reduction in execution time achieved with the wish jump/join/loop binary compared to the normal branch binary (row 1), the best-performing predicated code binary for the benchmark (row 2), and the best-performing binary (that does not contain wish branches) for the benchmark (row 3). Even if the compiler were able to choose and generate the best-performing binary for each benchmark, the wish jump/join/loop binary outperforms the best-performing binary for each benchmark by 5.1% on average, as shown in the third row.

5.3. Sensitivity to Microarchitectural Parameters

5.3.1. Effect of the Instruction Window Size

Figure 14 shows the normalized execution time of the wish jump/join/loop binaries on three different machines with 128, 256, and 512-entry instruction windows. The data shown in the left graph is averaged over all the benchmarks examined. The data in the right graph is averaged over all benchmarks except mcf. The execution time of each binary is normalized to the execution time of the normal branch binary on the machine with the corresponding instruction window size. Compared to the normal branch binaries, the wish jump/join/loop binaries improve the execution time by 11.4%, 13.0%, and 14.2% respectively on a 128, 256, and 512-entry window processor. Wish branches provide larger performance improvements on processors with larger instruction windows. This is due to the increased cost of branch mispredictions (due to the increased time to fill the instruction window after the pipeline is flushed) on machines with larger instruction windows. Wish loops are also more effective on larger windows, because, with a larger window, it is more likely that the front-end of the processor has already exited the loop when a mispredicted wish loop branch is resolved. This increases the likelihood of the late-exit case.

Table 5. Execution time reduction of the wish jump/join/loop binaries over the best-performing binaries on a per-benchmark basis (using the real confidence mechanism). DEF, MAX, BR (normal branch) indicate which binary is the best performing binary for a given benchmark.

		gzip	vpr	mcf	crafty	parser	gap	vortex	bzip2	twolf	AVG
1	% exec time reduction vs. normal branch binary	12.5%	36.3%	-1.5%	16.8%	23.1%	4.9%	3.2%	3.5%	29.8%	14.2%
2	% exec time reduction vs. the best predicated code binary for the benchmark	3.8% MAX	23.9% MAX	13.3% DEF	0.4% MAX	8.3% MAX	2.5% MAX	-4.3% DEF	-1.2% DEF	13.8% MAX	6.7%
3	% exec time reduction vs. the best non-wish-branch binary for the benchmark	3.8% MAX	23.9% MAX	-1.5% BR	0.4% MAX	8.3% MAX	2.5% MAX	-4.3% DEF	-1.2% DEF	13.8% MAX	5.1%

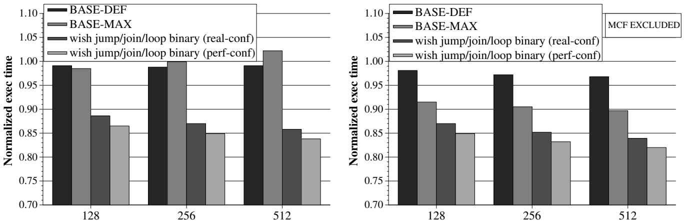


Figure 14. Effect of instruction window size on wish branch performance. The left graph shows the average execution time over all benchmarks, the right graph shows the average execution time over all benchmarks except mcf.

5.3.2. Effect of the Pipeline Depth Figure 15 shows the normalized execution time of the five binaries on three different 256-entry window processors with 10, 20, and 30 pipeline stages. Compared to the normal branch binaries, the wish jump/join/loop binaries improve the execution time by 8.0%, 11.0%, and 13.0% respectively on processors with 10, 20, and 30 pipeline stages. The performance benefits of wish branches increase as the pipeline depth increases, since the branch misprediction penalty is higher on processors with deeper pipelines. The wish jump/join/loop binaries always significantly outperform the normal branch and predicated code binaries for all pipeline depths and instruction window sizes examined.

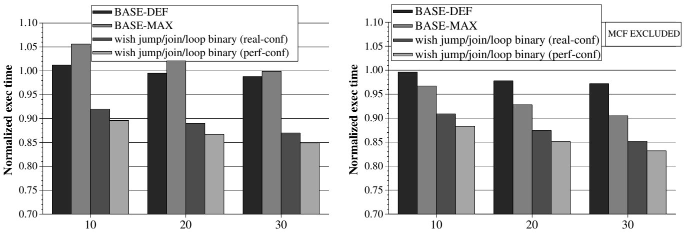


Figure 15. Effect of pipeline depth on wish branch performance.

5.3.3. Effect of the Mechanism Used to Support Predicated Execution Our baseline out-of-order processor uses C-style conditional expressions to handle predicated instructions as described in Section 2.1. We also implemented the select-μop mechanism proposed by Wang et al. [31] to quantify the benefits of wish branches on an out-of-order microarchitecture that uses a different technique to support predicated execution.

The advantage of the select-μop mechanism over the C-style conditional expressions is that it does not require the extra register read port and the extra input in the data-path to read and carry the old destination register value. Hence, the implementation cost of predicated execution is lower on a processor that supports predicated instructions using the select-μop mechanism. The select-μop also enables the execution of a predicated instruction before its source predicate value is ready, but the dependents of the predicated instruction still cannot be executed until the source predicate is resolved. Since dependent instructions cannot be executed, we found that a significant portion of the overhead of predicated execution still remains on a processor implementing the select-μop mechanism.

The disadvantage of the select-μop mechanism is that it requires additional μops to handle the processing of predicated instructions. Note that this is not the case in a processor that supports predicated instructions using C-style conditional expressions. Due to this additional μop overhead, the performance benefits of predicated code are lower on a processor that uses the select-μop mechanism than on a processor that uses C-style conditional expressions.

Figure 16 shows the normalized execution time of the predicated code, wish jump/join, and wish jump/join/loop binaries on a processor that supports predicated execution using the select-μop mechanism. With a real confidence estimator, the wish jump/join/loop binaries improve the average execution time by 11.0% compared to the normal branch binaries and by 14.0% compared to the best-performing (on average) predicated code binaries (BASE-DEF). On the processor that uses the select-μop mechanism, the overall performance improvement of wish branches over conditional branch prediction (11.0%) is smaller than it is on the processor that uses C-style conditional expressions (14.2%). This is due to the higher instruction overhead of the select-μop mechanism to support the predicated instructions. On the other hand, the overall performance improvement of wish branches over predicated execution (14.0%) is larger than it is on the processor that uses C-style conditional expressions (13.3%). Hence, the performance benefit of wish branches over predicated execution is larger when predicated execution has higher overhead.

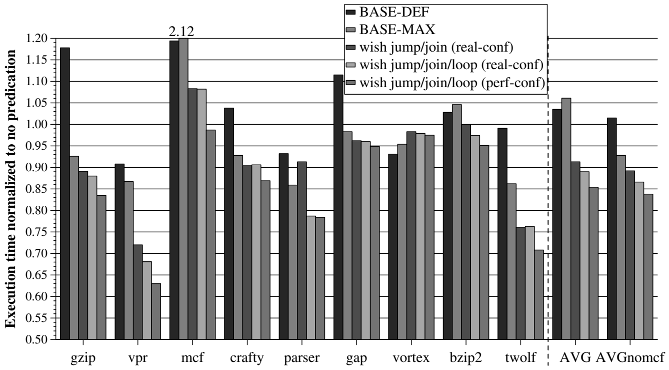


Figure 16. Performance of wish branches on an out-of-order processor that implements the select-μop mechanism

6. Related Work

6.1. Related Work on Predicated Execution

Several papers examined the impact of predicated execution on branch prediction and instruction-level parallelism. Pnevmatikatos and Sohi [23] showed that predicated execution can significantly increase a processor's ability to extract parallelism, but they also showed that predication results in the fetch and decode of a significant number of useless instructions. Mahlke et al. [19], Tyson [29], and Chang et al. [3] showed that predicated execution can eliminate a significant number of branch mispredictions and can therefore reduce the program execution time.

Choi et al. [5] examined the performance advantages and disadvantages of predicated execution on a real IA-64 implementation. They showed that even though predication can potentially remove 29% of the branch mispredictions in the SPEC CPU2000 INT benchmark suite, it results in only a 2% improvement in average execution time. For some benchmarks, a significant performance loss is observed with predicated execution. The performance loss in some benchmarks and the small performance gain in others are due to the overhead of predicated execution. Wish branches aim to reduce the overhead of predication by dynamically eliminating the useless predicated instructions.

Klauser et al. [14] proposed *dynamic hammock predication (DHP)*, which is a purely hardware mechanism that dynamically predicates hammock branches. Like wish branches, DHP enables the hardware to dynamically decide whether or not to use predication for a hammock branch. In contrast to wish branches, DHP is a purely hardware-based mechanism. DHP allows only simple control-flow graphs to be converted into predicated code. With wish branches, the compiler, which has a larger scope and more resources than the hardware, generates predicated code with a better understanding of the control-flow graph. Therefore, complex control-flow graphs as well as simple control-flow graphs can take advantage of wish branches. In summary, wish branches divide the work between the hardware and the compiler based on what each of them is better at: the compiler is better at analyzing the control flow comprehensively and the hardware is better at making decisions based on run-time behavior.

Chuang and Calder [7] proposed a hardware mechanism to predict *all* predicate values in order to overcome the register renaming problem in an out-of-order processor that implements predicated execution. Although they did not mention it, their mechanism can reduce the extra instruction overhead of predicated execution. With predicate prediction, instructions whose predicates are predicted false do not need to be executed, thus reducing the overhead of predicated execution—provided the prediction is correct. However, the processor still needs to fetch and decode all the predicated instructions. Wish branches can eliminate the fetch and decode of predicated instructions, as well as their execution. Also, *every* predicate is predicted with predicate prediction, which can result in performance loss for hard-to-predict predicates. Furthermore, wish branches can eliminate the misprediction penalty for backward (loop) branches, whereas conventional predication augmented with predicate prediction cannot.

6.2. Related Work on Control Flow Independence

Several hardware mechanisms have been proposed to exploit control flow independence [26] by reducing the branch misprediction penalty or improving parallelism [26, 6, 4]. These techniques aim to avoid flushing the processor pipeline when the processor is known to be at a control-independent point in the program at the time a branch misprediction is signalled. In contrast to wish branches, these mechanisms require a significant amount of hardware to exploit control flow independence [26]. Hardware is required to detect the reconvergent point dynamically, to remove the wrong-path instructions from the processor, to form correct data dependences for control-independent instructions, and to selectively re-schedule and re-execute the data-dependent instructions. Wish branches do not require such complicated hardware because they utilize predication.

6.3. Related Work on Multipath Execution

Several mechanisms were proposed to reduce the branch penalty by fetching and/or executing instructions from the multiple paths of the control flow. Eager execution [25] was proposed by Riseman and Foster. Dual-path fetch in IBM 360/91 [2] was a simple form of eager execution. Selective dual path execution [11], disjoint eager execution [30], and the PolyPath architecture [15] refined eager execution to reduce its implementation cost. These mechanisms are, in some respects, similar to the wish branch mechanism because they fetch and execute instructions from both paths of the control flow. However, in wish branch code, both control-flow paths after a conditional branch are already combined into one single path by predicating the code. No hardware support is needed to fetch from multiple paths. The hardware needs only to decide whether to fetch instructions from the taken path or from the not-taken path. In contrast, multipath execution requires extra hardware resources to fetch and execute from multiple control-flow paths.

7. Conclusions and Future Work

This paper proposes a new control-flow mechanism, called *wish branches*, to reduce the negative effects of predicated code and to obtain the best performance of predicated execution and branch prediction. We introduced and described the operation of three types of wish branches: wish jumps, wish joins, and wish loops. The major contributions of wish branches to the research in predicated execution and branch misprediction penalty reduction are:

1. Wish jumps and joins provide a mechanism to dynamically eliminate the overhead of predicated execution. These instructions allow the hardware to dynamically choose between using predicated execution versus conditional branch prediction for each dynamic instance of a branch based on the run-time confidence estimation of the branch's prediction.
2. Wish jumps and joins also allow the compiler to generate predicated code more aggressively and using simpler heuristics, since the "bad compile-time decisions" can be corrected at run-time. In previous research, a static branch instruction either remained as a conditional branch or was predicated for *all its dynamic instances*, based on less accurate compile-time information - if the compiler made a bad decision to predicate, there was no way to dynamically eliminate the overhead of the bad decision.
3. Wish loops provide a mechanism to exploit predicated execution to reduce the branch misprediction penalty for *backward* (loop) branches. In previous research, it was not possible to reduce the branch misprediction penalty for a backward branch by solely utilizing predicated execution.

Our results show that using wish branches improves the average execution time of nine SPEC INT 2000 benchmarks on an aggressive out-of-order superscalar processor by 14.2% compared to conditional branch prediction and by 13.3% compared to the best-performing predicated code binary.

We believe that the wish branch mechanism opens up opportunities that can make predicated execution more viable and effective in high performance processors. Since wish branches provide the hardware with the ability to dynamically eliminate the overhead of predicated execution, the compiler can generate predicated code more aggressively. Future work on wish branches can explore compiler algorithms and heuristics that can take advantage of wish branches. More accurate confidence estimation mechanisms are also interesting to investigate since they would increase the performance benefits of wish branches.

Acknowledgments

Special thanks to David Armstrong for the initial simulator work and Robert Cohn for supporting PIN. We thank Hsien-Hsin S. Lee and Georgia Tech College of Computing and HP TestDrive for providing the use of IA-64 machines. We thank Roy Ju, Derek Chiou, Santhosh Srinath, Francis Tseng, Mary Brown, David Thompson, Chang Joo Lee and other members of the HPS research group and the anonymous reviewers for their comments and suggestions. We gratefully acknowledge the commitment of the Cockrell Foundation, Intel Corporation and The Advanced Technology Program of the Texas Higher Education Coordinating Board for supporting our research at The University of Texas at Austin.

References

[1] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *POPL-10*, 1983.

[2] D. Anderson, F. Sparacio, and R. Tomasulo. The IBM system/360 model 91: Machine philosophy and instruction-handling. *IBM Journal of Research and Development*, 11(1):8–24, Jan. 1967.

[3] P.-Y. Chang, E. Hao, T.-Y. Yeh, and Y. N. Patt. Using predicated execution to improve the performance of a dynamically-scheduled machine with speculative execution. In *PACT-1995*, 1995.

[4] C.-Y. Cher and T. N. Vijaykumar. Skipper: a microarchitecture for exploiting control-flow independence. In *MICRO-34*, 2001.

[5] Y. Choi, A. Knies, L. Gerke, and T.-F. Ngai. The impact of if-conversion and branch prediction on program execution on the Intel Itanium processor. In *MICRO-34*, 2001.

[6] Y. Chou, J. Fung, and J. P. Shen. Reducing branch misprediction penalties via dynamic control independence detection. In *ICS-99*, 1999.

[7] W. Chuang and B. Calder. Predicate prediction for efficient out-of-order execution. In *ICS-03*, 2003.

[8] Compaq Computer Corporation. *Alpha 21264 Microprocessor Hardware Reference Manual*, 1999.

[9] A. Darsch and A. Seznec. *IATO, The IAOO Toolkit*. IRISA. http://www.irisa.fr/caps/projects/ArchiCompil/iato/.

[10] M. R. de Alba and D. R. Kaeli. Runtime predictability of loops. In *IEEE 4th Annual Workshop on Workload Characterization*, 2001.

[11] T. Heil and J. E. Smith. Selective dual path execution. Technical report, University of Wisconsin-Madison, Nov. 1996.

[12] Intel Corporation. *IA-64 Intel Itanium Architecture Software Developer's Manual Volume 3: Instruction Set Reference*, 2002.

[13] E. Jacobsen, E. Rotenberg, and J. E. Smith. Assigning confidence to conditional branch predictions. In *MICRO-29*, 1996.

[14] A. Klauser, T. Austin, D. Grunwald, and B. Calder. Dynamic hammock predication for non-predicated instruction set architectures. In *PACT-1998*, 1998.

[15] A. Klauser, A. Paithankar, and D. Grunwald. Selective eager execution on the polypath architecture. In *ISCA-25*, 1998.

[16] A. KleinOsowski and D. J. Lilja. Minnespec: A new SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1, June 2002.

[17] Y. Liu, Z. Zhang, R. Qiao, and R. Ju. A region-based compilation infrastructure. In *INTERACT-7*, 2003.

[18] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[19] 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 *MICRO-27*, 1994.

[20] S. Mantripragada and A. Nicolau. Using profiling to reduce branch misprediction costs on a dynamically scheduled processor. In *ICS-2000*, 2000.

[21] S. McFarling. Combining branch predictors. Technical Report TN-36, Digital Western Research Laboratory, June 1993.

[22] ORC. Open research compiler for Itanium processor family. http://ipf-orc.sourceforge.net/.

[23] D. N. Pnevmatikatos and G. S. Sohi. Guarded execution and dynamic branch prediction in dynamic ILP processors. In *ISCA-21*, 1994.

[24] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle. The Cydra 5 departmental supercomputer. *IEEE Computer*, 22:12–35, Jan. 1989.

[25] E. M. Riseman and C. C. Foster. The inhibition of potential parallelism by conditional jumps. *IEEE Transactions on Computers*, C-21(12):1405–1411, 1972.

[26] E. Rotenberg, Q. Jacobson, and J. E. Smith. A study of control independence in superscalar processors. In *HPCA-5*, 1999.

[27] T. Sherwood and B. Calder. Loop termination prediction. In *HiPC-3*, 2000.

[28] E. Sprangle and Y. Patt. Facilitating superscalar processing via a combined static/dynamic register renaming scheme. In *MICRO-27*, 1994.

[29] G. S. Tyson. The effects of predication on branch prediction. In *MICRO-27*, 1994.

[30] A. Uht and V. Sindagi. Disjoint eager execution: An optimal form of speculative execution. In *ISCA-22*, 1995.

[31] P. H. Wang, H. Wang, R. M. Kling, K. Ramakrishnan, and J. P. Shen. Register renaming and scheduling for dynamic execution of predicated code. In *HPCA-7*, 2001.

[32] T.-Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. In *ISCA-19*, 1992.