Wish Branch: A New Control Flow Instruction Combining Conditional Branching and Predicated Execution

Hyesoon Kim Onur Mutlu Jared Stark ‡ David N. Armstrong Yale N. Patt

High Performance Systems Group
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, Texas 78712-0240

‡Microprocessor Research Labs
Intel Corporation
Hillsboro, OR 97124

TR-HPS-2005-002
February 2005

This page is intentionally left blank.

Wish Branch: A New Control Flow Instruction Combining Conditional Branching and Predicated Execution

Hyesoon Kim Onur Mutlu Jared Stark ‡ David N. Armstrong Yale N. Patt

ECE Department
The University of Texas at Austin
{hyesoon,onur,dna,patt}@ece.utexas.edu

‡Microprocessor Research Labs
Intel Corporation
jared.w.stark@intel.com

Abstract

As processor pipelines get deeper and wider and instruction windows get larger, the branch misprediction penalty increases. Predication has been used to reduce the number of branch mispredictions by eliminating hard-to-predict branches. However, with predication, the processor is guaranteed to fetch and possibly execute useless instructions, which sometimes offsets the performance advantage of having fewer mispredictions. Also, predication does not eliminate the misprediction penalty due to backward loop branches.

This paper introduces a new type of branch called a wish branch. Wish branches combine the strengths of traditional conditional branches and predication, allowing instructions to be skipped over as with traditional branches, but also avoiding pipeline flushes due to mispredictions as with predication. Unlike traditional conditional branches, on a wish branch misprediction, the pipeline does not (always) need to be flushed. And, unlike predication, a wish branch (sometimes) avoids fetching from both paths of the control flow. This paper also describes a type of wish branch instruction, called a wish loop, which reduces the branch misprediction penalty for backward loop branches.

We describe the software and hardware support required to generate and utilize wish branches. We demonstrate that wish branches can decrease the execution times of SPEC2000 integer benchmarks by 7.8% (up to 21%) compared to traditional conditional branches and by 3% (up to 11%) compared to predicated execution. We also describe several simple hardware optimizations for exploiting and increasing the benefits of wish branches.

1. Introduction

As processor pipelines get deeper and wider and instruction windows get larger, the branch misprediction penalty increases. Predicated execution is used to eliminate hard-to-predict branches by converting the control dependencies to data dependencies [1, 14]. Using predicated execution, a processor fetches and executes more instructions but it does not incur the branch misprediction penalty for the eliminated branches.[1] However, even with code carefully predicated by a state-of-the-art compiler, a processor implementing predicated execution sometimes loses performance because of the overhead due to the unnecessary instructions [6]. If the compiler predicates a branch because profile information indicates the branch is hard-to-predict, but at run-time the branch is easy-to-predict because the profile and actual input sets are different or the branch is predictable in some program phases but not in others, predicated execution hurts performance.

[1] Depending on the microarchitecture design, a predicated instruction with a *false* predicate may be fetched, decoded, and renamed, but not executed in a functional unit.

We introduce a new control flow instruction, called a *wish branch*. With wish branches, we can combine normal conditional branching with predicated execution, providing the benefits of predicated execution without its wasted fetch and execution bandwidth. Wish branches aim to reduce the branch misprediction penalty by using predicated execution only when it increases performance. The decision of when to use predicated execution is made during run-time using a branch predictor and, optionally, a confidence estimator. While in some run-time scenarios normal branches perform better than predicated execution, predicated execution performs better in others. Wish branches aim to get the better performance of the two under all scenarios.

A wish branch looks like a normal branch but the code on the fall-through path between the branch and the target is predicated. A forward wish branch is called a *wish jump*. When the processor fetches the wish jump, it predicts the direction of the wish jump using a branch predictor, just like it does for a normal branch. If the wish jump is predicted not-taken, the processor executes the predicated code. But if it is mispredicted, the pipeline does not need to be flushed since the fall-through path is predicated. If the wish jump is predicted taken, the processor executes the normal branch code. If this prediction is correct, the extra useless instructions in the predicated code are not fetched. Hence, a wish jump can obtain the better performance of a normal branch and predicated execution. To increase the benefit of wish jumps, wish jumps can be used with a confidence estimator. When the confidence estimator predicts that a wish jump might be mispredicted, the hardware performs predicated execution. Thus, the wish jump mechanism gives the hardware the option to dynamically decide whether or not to use predicated execution.

Previous research shows that backward loop branches cannot be directly eliminated using predication [1], which reduces predicated execution's opportunity to remove the misprediction penalty for a large percentage of branches [6]. However, a backward loop branch can be converted to a wish branch instruction, which we call a *wish loop*. We show that the wish loop instruction can reduce the branch misprediction penalty by exploiting the benefits of predicated execution for backward branches. To use the wish loop, the compiler predicates the body of the loop using the loop branch condition as the predicate. When the wish loop is mispredicted, the processor doesn't need to flush the pipeline because the body of the loop is predicated.

1.1. Contributions

This paper makes the following contributions to the research in reducing the branch penalty in microprocessors:

1. We propose a novel control flow instruction, called a *wish branch*, that combines the strengths of predicated execution and conditional branch prediction. This instruction:
   (a) provides the hardware with a *choice* to use branch prediction *or* predicated execution for each dynamic instance of a branch. In previous research, a static branch instruction either remained as a conditional branch or was predicated for all its dynamic instances, based on compile-time information. Exploiting the choice provided by a wish branch, the hardware can decide whether or not to use predicated execution based on more accurate run-time information.
   (b) provides a mechanism to exploit predicated execution to reduce the branch misprediction penalty for *backward* branches. In previous research, it was not possible to reduce the branch misprediction penalty for a backward branch by solely utilizing predicated execution.
2. We develop a compiler algorithm to generate wish branches and describe compiler optimizations that increase the performance benefits of wish branches.

3. We describe the basic hardware mechanisms to support wish branches and propose hardware enhancements, including a *wish loop branch predictor*, that increase performance by exploiting the characteristics specific to wish branches.
4. We demonstrate the use, benefits, and shortcomings of wish branches by implementing the wish branch code generation algorithm in a state-of-the-art compiler and simulating the generated code in a cycle-accurate, state-of-the-art out-of-order processor model implementing a fully-predicated ISA. Our evaluation reveals that:
   (a) Wish branches outperform both predicated execution and conditional branch prediction.
   (b) Wish branches reduce the negative effects of predicated execution when predicated execution results in performance degradation.
5. We describe the shortcomings of wish branches and describe future research directions on how to reduce these shortcomings.

2. Wish Branches

Conventional branches have compulsory branching behavior: if the branch predicate is false, the processor *must* fall through; if the predicate is true, the processor *must* take the branch. Wish branches have optional –but still desired– branching behavior for one of the predicate values. For example, if the branch predicate is false, the processor *must* fall through. However, if the predicate is true, the compiler desires, or wishes, that the processor take the branch. But if the processor doesn't take the branch, it is OK, because the compiler has arranged for the code to produce the correct outcome regardless of whether the processor takes the branch.

In this section, we explain the behavior of wish branches and how wish branches are different from normal branches and predicated execution. We introduce and describe three different wish branch instructions: (1) wish jump (Section 2.1), (2) wish join (Section 2.2), and (3) wish loop (Section 2.3).

2.1. Wish Jumps

Figure 1 shows a source code example and the corresponding control flow graphs and assembly code for: (a) a normal branch, (b) predicated execution, (c) a wish jump, and (d) a wish jump with a wish join. The main difference between the wish jump code and the normal branch code is that the instructions in basic blocks B and C are predicated in the wish jump code (Figure 1c), but they are not predicated in the normal branch code (Figure 1a). The other difference is that the normal branch code has an unconditional branch at the end of block B, but the wish jump code doesn't have one. The difference between the wish jump code and the predicated code (Figure 1b) is that the wish jump code has a branch whose target is block C, but the predicated code doesn't have any branches.

In the normal branch code, only block B or block C is executed depending on the branch condition. Predicated execution always executes both blocks B and C. In the wish jump code, if the condition of the wish jump is true (i.e., it is taken), only block C is executed. If the condition of the wish jump is false, both blocks B and C are executed. Hence, wish jump code is just like normal branch code when the condition of the wish jump is true, and it is just like predicated code when the condition of the wish jump is false.

When the processor fetches the wish jump instruction, it predicts the direction of the wish jump using a branch predictor, just like it does for a normal conditional branch. There are four cases based on the predicted direction and the actual direction: (1) Predicted Taken and the Actual direction is Taken (PTAT), (2) Predicted Taken and the Actual direction is Not-taken (PTAN), (3) Predicted Not-taken and the Actual direction is Taken (PNAT), and (4) Predicted Not-taken and the Actual direction is Not-taken (PNAN).

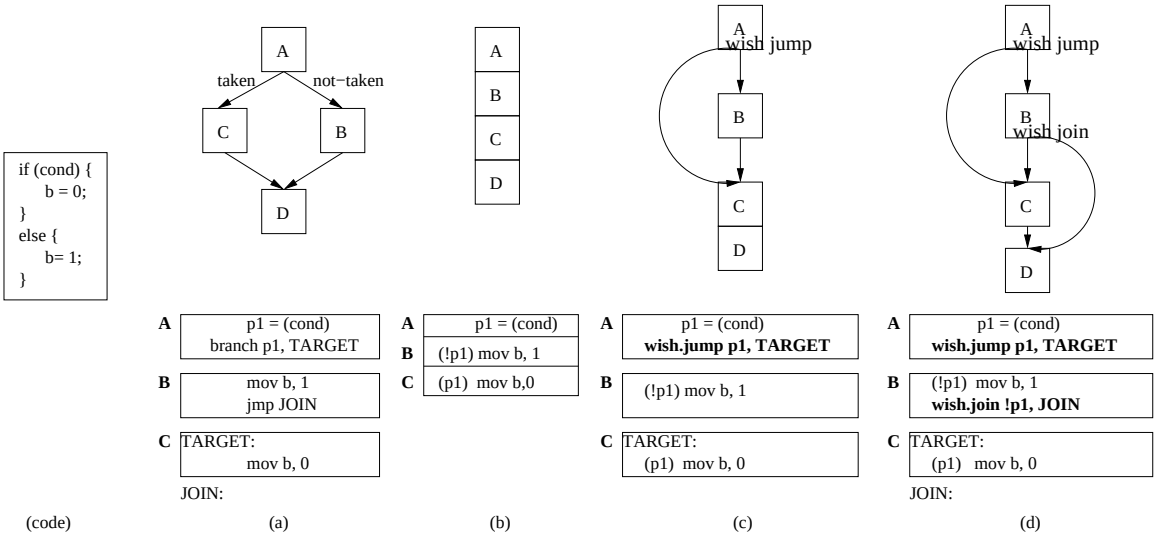


Figure 1. Source code (code); the corresponding control flow graph and assembly code for (a) normal branch code (b) predicated code (c) wish jump code (d) wish jump and wish join code

For cases PTAT and PNAN, the wish jump is correctly predicted. For cases PTAN and PNAT, the wish jump is mispredicted. With a normal branch, when the branch is mispredicted, the processor always flushes the pipeline. With a wish jump, PTAN results in a pipeline flush, but PNAT does not. For the PNAT case, the processor has already fetched blocks B and C whose instructions are predicated. The instructions in block B become NOPs after the predicate value is available, eliminating the need for a pipeline flush. For the PTAN case, the processor has not fetched block B, which actually needs to be executed, so it flushes the pipeline just like in case of a normal branch and it fetches both blocks B and C after misprediction recovery.

Table 1 summarizes the processor's action for each of the four cases and compares the cost and benefit of wish jumps to normal branches and predicated execution. A wish jump performs better than a normal branch for the PNAT case, because the processor doesn't flush the pipeline even though the wish jump is mispredicted. However, the processor fetches and executes more instructions for the PTAN, PNAT, and PNAN cases compared to the normal branch. For the PTAT case, the wish jump behaves exactly the same as a normal branch.[2] A wish jump performs better than predicated execution for the PTAT case, because the wish jump doesn't need to fetch and execute block B. However, the wish jump results in a branch misprediction for the PTAN case, which is worse than predicated execution. In PNAT and PNAN, the wish jump and predicated execution perform the same except that the wish jump results in the execution of one more instruction, which is the wish jump itself.

Table 1. Cost-benefit comparison between wish jump and normal branch, predicated execution

Case	Processor's action for wish jump code	Comparison to normal branch code	Comparison to predicated code
PTAT	fetch and execute block C	same **(=)**	no fetch and execution of B **(+)**
PTAN	pipeline flush and fetch, execute block B and C (C becomes nop)	extra fetch and execution of C **(-)**	pipeline flush penalty **(- -)**
PNAT	fetch, execute block B and C (B becomes nop)	no flush penalty but fetch and execution of B **(++)**	similar (one extra instruction) **(≅)**
PNAN	fetch, execute block B and C (C becomes nop)	extra fetch and execution of C. execution delay of B **(-)**	similar (one extra instruction) **(≅)**

[2] See Section 5.3.1 for further information on the PTAT case.

How wish jumps affect the overall performance depends on the frequency and relative cycle impact of each of the four cases. Section 3.1 explains how a compiler uses these cases to decide when a wish jump is needed. Section 5 shows the frequency of each case and the net performance impact of wish jumps.

2.2. Wish Joins

In Figure 1c, even when the wish jump is correctly predicted to be not-taken (PNAN), the processor needs to fetch both blocks B and C. To avoid the execution of basic block C in this case, a wish jump can be used with another conditional branch instruction at the end of block B, as shown in Figure 1d. This instruction is called a *wish join*.

When a wish join is taken, the wish jump and wish join together behave exactly the same as a normal branch. When a wish join is not-taken, the wish jump and wish join together behave like predicated execution. The main benefit of wish jumps with wish joins is the extra flexibility they give to the processor in deciding whether or not to use predicated execution. A wish join is used with a confidence estimator for the wish jump. When the wish jump is predicted, a confidence estimation is provided for this prediction. If the wish jump prediction has low confidence, both the wish jump and the corresponding wish join are predicted not-taken (the same as predicated execution). If the wish jump prediction has high confidence, the wish jump is predicted according to the branch predictor and the wish join is predicted taken (the same as traditional conditional branch prediction). Hence, this algorithm results in the use of predicated execution for low-confidence branches and the use of branch prediction for high-confidence branches, where the confidence of a branch is determined dynamically.

Based on the prediction for the wish join, the four cases in Section 2.1 become six cases. PNAN becomes either PNAN-JT (Join is predicted Taken) or PNAN-JN (Join is predicted Not-taken). PNAT becomes either PNAT-JT or PNAT-JN. To distinguish the wish jump behavior in the previous section from the wish join behavior in this section, we relabel the PTAT case as PTAT-J and the PTAN case as PTAN-J.[3]

When the wish jump is correctly predicted not-taken and the wish join is predicted taken (PNAN-JT), the processor avoids fetching block C, which is the goal of the wish join, and an improvement over the wish jump case PNAN. When the wish jump is incorrectly predicted not-taken and the wish join is predicted taken (PNAT-JT), the processor needs to flush the pipeline, a loss compared to the wish jump case PNAT. And when the wish join is predicted not-taken (PNAN-JN or PNAT-JN), the performance of using the wish jump with the wish join is the same as the performance of using the wish jump by itself, except that the use of the wish join adds one more instruction (the wish join) to the code. Finally, if the wish jump is incorrectly predicted taken (PTAN-J), after the wish jump is resolved and misprediction recovery is complete, the processor doesn't need to fetch block C, an improvement over the wish jump case PTAN. Table 2 summarizes the processor's action for all six cases and compares the cost and benefit of a wish jump with a wish join to a normal branch and predicated execution.

Table 2. Cost-benefit comparison between wish jump with wish join and normal branch, predicated execution

Case	Processor's action for wish jump w/ wish join	Comparison to normal branch code	Comparison to predicated code
PTAT-J	fetch and execute C	same **(=)**	no fetch and execution of B **(+)**
PTAN-J	pipeline flush and fetch and execute B	same **(=)**	pipeline flush, but no fetch and exec. of C after flush **(- -)**
PNAT-JN	fetch and execute B and C (B becomes nop)	no flush penalty but fetch and execution of B **(++)**	similar (two extra instructions) **(≅)**
PNAT-JT	pipeline flush and fetch and execute C	same **(=)**	pipeline flush, but no fetch and exec. of B after flush **(- -)**
PNAN-JN	fetch and execute B and C (C becomes nop)	extra fetch and execution of C. execution delay of B **(-)**	similar (two extra instructions) **(≅)**
PNAN-JT	fetch and execute B	same **(=)**	no fetch and execution of C **(+)**

[3]Note that the actual direction of the wish join is opposite the actual direction of the wish jump. Also note that if the wish jump is predicted taken, the wish join is not fetched. That's why we do not distinguish between PTAT-JT and PTAT-JN, since they are the same case, which we call PTAT-J. The same is true for PTAN-J.

2.3. Wish Loops

We have explained wish branches only for forward branches. A wish branch can also be used for a backward branch. We call this a *wish loop* instruction. Figure 2 contains the source code for a simple loop body and the corresponding control-flow graph 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 utilizes predication to reduce the branch misprediction penalty of a backward branch without eliminating the branch.

The main difference between the normal branch code (Figure 2a) and the wish loop code (Figure 2b) is that the instructions in block X (i.e., the loop body) are predicated using the loop branch condition in the wish loop code. Wish loop code also contains an extra instruction in the loop header for the initialization of the predicate.

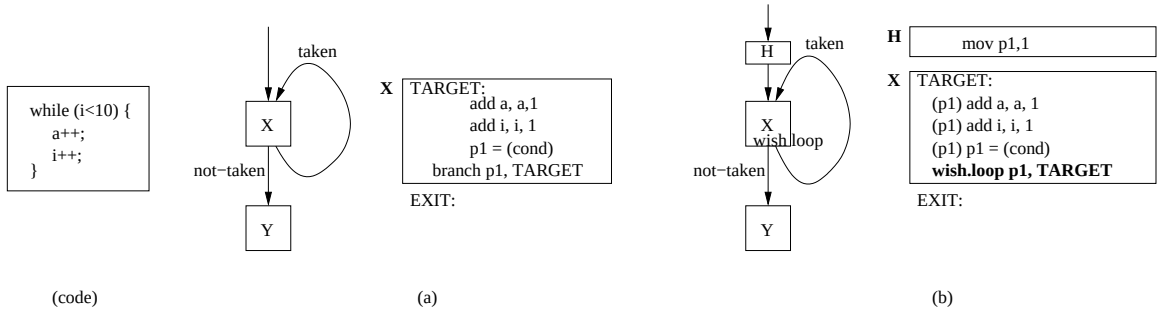


Figure 2. Source code (code); the corresponding control flow graph and assembly code for (a) normal backward branch code (b) wish loop code

With wish loops, there are three misprediction cases: (1) *early-exit*: the loop is iterated fewer times than it is supposed to be, (2) *late-exit*: the loop is iterated only a few more times by the processor front end than it is supposed to be and it is already exited when the wish loop misprediction is signalled, (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's iterated more times than needed).

For example, say a loop needs to be iterated 3 times. The correct loop branch direction is TTN (taken, taken, not-taken) for the three iterations and the front end needs to fetch blocks $X_1X_2X_3Y$. 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 has already fetched blocks X_1X_2Y. One example of the late-exit case is when the predictions for the loop branch are TTTTN and the front end has already fetched blocks $X_1X_2X_3X_4X_5Y$. For the no-exit case, the predictions for the loop branch are TTTTT...T and the front end has already fetched blocks X_1,X_2,X_3,X_4,X_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 branch. In the late-exit case, the instructions in blocks X_4 and X_5 become NOPs because the predicate value p1 is 0 for the extra iterations 4 and 5, and the processor has already fetched block Y. Therefore, the processor doesn't need to flush the pipeline. With the late-exit case, the wish loop performs better than a normal backward branch. In this case, as long as the processor front end has already fetched block Y, the pipeline does not need to be flushed (regardless of how many extra loop iterations are fetched). Hence, the wish loop saves part of the branch misprediction penalty. The fewer the number of extra loop iterations fetched, the larger the savings in the branch misprediction penalty.

In the no-exit case, the front end has never fetched block Y. Therefore, the processor flushes the pipeline and restarts with fetching block Y, similar to the action taken for a normal mispredicted branch. It is possible to not flush the pipeline

in this case (since the extra iterations that have filled the pipeline are already predicated and will turn into NOPs), but not flushing the pipeline does not result in any reduction in the branch misprediction penalty. Furthermore, not flushing the pipeline requires the useless iterations to be drained from the pipeline, which may reduce performance by holding on to resources needed by incoming correct-path instructions.

We note that when the wish loop is correctly predicted, the wish loop code performs worse than the normal branch code. First, the wish loop code contains one extra instruction in the loop header. Second, and more importantly, because all the instructions in the loop body (block X) are predicated, the instructions cannot be executed until the predicate value is available, which results in an execution delay that can reduce performance compared to normal branch code where the loop exit is correctly predicted. Due to these overheads, a good compiler should selectively convert backward loop branches to wish loops using cost-benefit analysis or heuristics.

Table 3 summarizes the processor's action for the case where the loop exit is correctly predicted and the three misprediction cases and compares the cost and benefit of wish loop code to normal branch code. We expect wish loops to do well in integer benchmarks where loops iterate a variable number of times in an unpredictable manner [11] that cannot be captured by a loop branch predictor [34]. As wish loops reduce the misprediction penalty for the late-exit case, a specialized wish loop predictor can be designed to predict wish loops. 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 wish loop that is hard to predict. We describe the design of such a loop predictor in Section 5.4.

Table 3. Cost-benefit comparison between wish loop code and normal branch code

Case	Processor's action for wish loop code	Comparison to normal branch code
correct-prediction	fetch and execute block X exactly as needed	execution delay (and one extra instruction) **(-)**
early-exit	pipeline flush and fetch, execute block X as needed and then block Y	execution delay (and one extra instruction) **(-)**
late-exit	fetch and execute block X more than needed and fetch block Y	exec. delay and extra iterations, but no flush penalty (and one extra inst.) **(++)**
no-exit	pipeline flush and fetch, execute block Y	execution delay (and one extra instruction) **(-)**

2.4. Why Are Wish Branches a Good Idea?

1. A wish jump eliminates the negative effects (extra instruction fetch and execution overhead) of predicated execution, if the predicated branch turns out to be easy-to-predict at run-time.
2. Wish branches provide a choice to the hardware: the choice of *whether or not to use predicated execution*. This choice does not exist for a traditional conditional branch or predicated execution. With a wish jump and a wish join, wish branch code can behave exactly the same as a normal branch *or* predicated code, depending on the direction of the wish branch. The processor can decide to use predicated execution for a low-confidence branch and use branch prediction for a high-confidence branch, where the confidence of a branch is determined dynamically. Hence, wish branches allow the processor to obtain the better performance of conditional branch prediction and predicated execution by utilizing run-time information about branch behavior (in contrast to the less accurate compile-time information utilized by the compiler for predicated execution).
3. With the wish loop instruction, a backward branch can utilize the benefits of predicated execution. Like wish jumps, wish loops can also reduce the branch misprediction penalty by sometimes avoiding the flush of the pipeline.

2.5. ISA Support

Figure 3 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 btype is 0, the branch is a normal branch and if btype is 1, the branch is a wish branch. If wtype is 0, the wish branch is a wish jump; if wtype is 1, the wish branch is a wish loop; if wtype is 2, the wish branch is a wish join. If the current ISA already has hint bits for the conditional branch instruction, like the IA-64 [15], wish branches can be implemented using the hint bit fields without modifying the ISA. 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 then run correctly on existing processors without wish branch support.

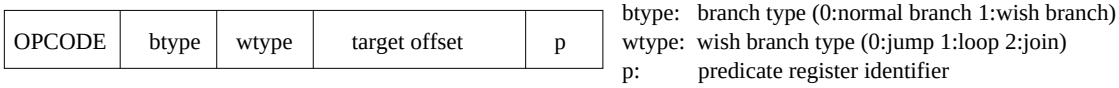


Figure 3. A possible instruction format for the wish branch

2.6. Hardware Support

Aside from the hardware required to support predicated execution, wish branches require hardware support in the instruction decoding logic and the branch misprediction detection/recovery module. When a wish jump is mispredicted, if the predicted branch direction was not-taken, the processor does not flush the pipeline. This corresponds to the PNAT case in Section 2.1. If a wish join is also used, the processor also needs to check the prediction for the wish join. If the wish join is also predicted to be not-taken, the pipeline does not need to be flushed, corresponding to the PNAT-JN case in Section 2.2.

To support wish loop misprediction recovery, 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 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,[4] and the processor flushes the pipeline. Note that to keep the hardware simple we don't support nested wish loops.

3. Compiler Algorithm for Converting Conditional Branches to Wish Branches

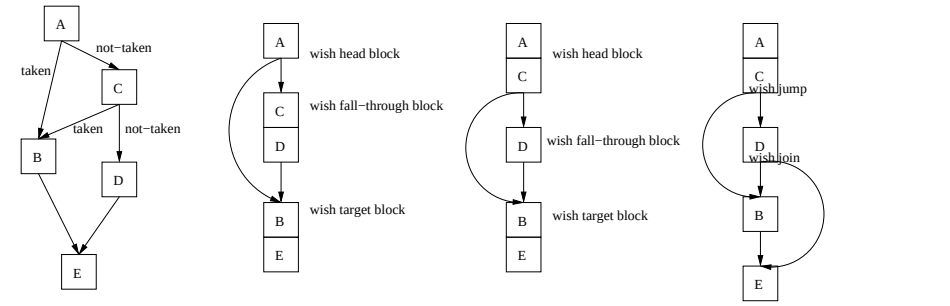
Conversion of branches to wish jumps and wish joins occurs during the if-conversion phase of the compilation process. During this phase, the compiler decides whether a branch is converted to a wish jump, a wish jump and wish join pair, is predicated, or stays unchanged. Conversion of backward branches to wish loops occurs after the if-conversion phase.

3.1. Wish Jump Conversion

If the compiler decides to convert a branch to a wish jump, the original control flow graph is converted to a new control flow graph, which we call the *wish control flow graph*. An example control flow graph is shown in Figure 4a and

[4]Note that 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. We found that such unnecessary flushes occur rarely.

its corresponding wish control flow graph is shown in Figure 4b. A wish control flow graph consists of three blocks: a *wish head block*, a *wish fall-through block*, and a *wish target block*.



(a) normal branch code (b) wish jump code (A has wish jump) (c) wish jump code (C has wish jump) (d) wish jump with wish join (C has wish jump)

Figure 4. Wish control flow graph examples

3.1.1. First step in conversion: Selection of wish jump instructions. The compiler first selects wish jump candidate instructions based on cost-benefit analysis. To do so, it scans all the basic blocks in a given region and selects the basic blocks which can be predicated [25]. Only branches that can be predicated can be potentially converted into wish jumps.[5] Hence, only a basic block that can be predicated can be part of the wish fall-through block or wish target block. Whether or not a basic block becomes part of the wish fall-through block or the wish target block is determined through control flow graph analysis. If a block can be part of either the wish fall-through block or the wish target block, then the block is included in the wish fall-through block. For example, due to the control flow characteristics in Figure 4a, basic block D can be part of the wish fall-through block or the wish target block in Figure 4b. Our compiler algorithm makes such blocks part of the wish fall-through block, because a larger fall-through block size reduces the number of useless instructions executed when the wish jump is taken. After identifying the control-flow graph structure of the wish jump code, the compiler estimates the execution time of the wish jump code. Execution time of the normal branch code and the predicated code for the same portion of the control flow graph are also estimated. Equations used for execution time estimates for these cases are as follows:

$$Exec\,time\,of\,normal\,branch\,code = exec\_T * [P(PTAT) + P(PNAT)] + exec\_N * [P(PNAN) + P(PTAN)] + misp\_penalty * [P(PTAN) + P(PNAT)]$$

$$Execution\,time\,of\,predicated\,code = exec\_pred$$

$$Exec\,time\,of\,wish\,jump\,code = exec\_T * P(PTAT) + exec\_pred * [P(PNAN) + P(PNAT) + P(PTAN)] + misp\_penalty * P(PTAN)$$

exec_T : execution time of the code when the branch under consideration is taken
exec_N: execution time of the code when the branch under consideration is not taken
exec_pred : execution time of the predicated code (without considering the predicate dependency)
misp_penalty: machine-specific branch misprediction penalty
P(case) : The probability of the case; e.g., P(PNAT) is the probability of case PNAT.

The probability of each case (PTAT, PTAN, PNAT, PNAN) is estimated by profiling or using compiler heuristics. Execution times (exec_T, exec_N, exec_pred) are estimated with dependency height and resource usage analysis. If the execution time estimate for the wish jump code is the least among the three execution time estimates, the branch is converted to a wish jump. If the execution time estimate for the predicated code is the least, then the code is predicated.

When there are several wish jump candidates in a given selection, the compiler performs cost-analysis for each wish jump candidate. For example, in Figure 4a, either the branch at the end of basic block A or the branch at the end of basic block C can be converted to wish jump. Figure 4b shows the wish control flow graph when the branch at the end of A is converted and Figure 4c shows the wish control flow graph when the branch at the end of C is converted.

[5] We use the ORC compiler's baseline algorithm [21, 25, 24] to determine the branches that can be predicated.

3.1.2. Second step in conversion: Wish block conversion and predication. After finishing the wish jump selection, the compiler performs wish block formation. Wish target blocks and wish fall-through blocks are predicated. Multiple basic blocks in each wish block are combined to form a single basic block (e.g., in Figure 4b, blocks B and E form a single basic block when combined). The selected wish jump is marked to be a wish jump and all unnecessary branches are eliminated. At the code emit stage, the compiler generates code such that a wish head block, the corresponding wish fall-through block and the wish target block are laid out next to each other.

3.2. Wish Join Insertion

A wish join is inserted at the end of a wish fall-through block after wish block formation. Because the original control flow graph has already been converted to a wish control flow graph, the potential for inserting a wish join instruction is limited. The potential for wish join insertion is also reduced by the existence of *if* statements without corresponding *else* statements in the source code. If the wish target block might need to be executed when the wish jump is not taken, a wish join instruction is not inserted. For example, in Figure 4b, block B might need to be executed even if the wish jump is not taken. Our compiler algorithm does not insert the wish join instruction in this case. In Figure 4c, block B never needs to be executed if the wish jump is not taken, so a wish join instruction is inserted. Figure 4d shows the control flow graph after a wish join is inserted in Figure 4c.

The compiler examines the instructions in the wish target block to determine whether or not to insert a wish join. If there are instructions which always become NOPs when the wish jump condition is false, it inserts a wish join instruction. The target of the wish join instruction is the first instruction which might need to be executed when the wish jump condition is false.

3.3. Wish Loop Conversion

Conversion of backward branches to wish loops occurs after the if-conversion phase. First, the compiler identifies which branches need to be converted to wish loops. All backward branches can potentially be wish loops. To simplify the hardware needed to support wish loops, we do not allow nested wish loops. The loop body is also not allowed to contain any loop exit branches or procedure calls. As discussed in Section 2.3, due to the execution delay resulting from the predication of the loop body, wish loops need to be selected using cost-benefit analysis. Equations used for execution time estimates are as follows:

$$Iteration\ exec\ time\ (normal\ loop\ code) = exec\_loop + misp\_penalty * [P(early\_exit) + P(late\_exit) + P(no\_exit)]$$

$$Iteration\ exec\ time\ (wish\ loop\ code) = exec\_loop + misp\_penalty * [P(early\_exit) + P(no\_exit)] + exec\_delay + late\_exit\_penalty * P(late\_exit) + 1/N$$

exec_loop: average execution time of the loop body (one iteration) without considering the predicate dependency
exec_delay: execution delay of one iteration due to the predicate dependency
late_exit_penalty: average number of cycles the processor spends for extra loop iterations in the late-exit case
1/N: per-iteration execution time for one extra instruction added to the loop header (N is the average number of loop iterations)

Since it is hard to estimate the exec_delay for an out-of-order processor, instead of cost-benefit analysis, our compiler uses a simple heuristic that considers the size of the loop body, which affects the exec_delay: the compiler converts a branch to a wish loop only if the number of static instructions in the loop body is less than or equal to N. We explore the performance impact of the threshold N in Section 5.2.

After deciding that a backward branch will be converted to a wish loop, the compiler inserts a header block before the loop body in the control flow graph. A predicate initialization instruction is inserted in this header block. All the instructions in the loop body are then predicated with the loop branch condition.

4. Methodology

Figure 5 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 and used the ORC compiler [25] 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 [26] 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.

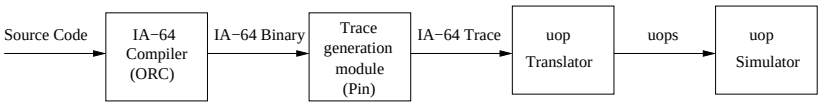


Figure 5. Simulation infrastructure

4.1. Simulator

Experiments were run using a cycle-accurate simulator which was derived from an in-house Alpha ISA simulator. Our baseline processor is an aggressive superscalar, out-of-order processor. Table 4 describes our baseline micro-architecture. Because a less accurate branch predictor would provide more opportunity for wish branches, a large and accurate hybrid branch predictor is used in our experiments to avoid inflating the impact of wish branches.

Table 4. Baseline processor configuration

Front End	64KB, 4-way instruction cache with 2-cycle latency; 8-wide fetch/decode/rename
Branch Predictors	64K-entry gshare, 64K-entry PAs hybrid with 64K-entry selector; 4K-entry branch target buffer; 64-entry return address stack; 64K-entry target cache (for indirect branches); minimum branch misprediction penalty is 30 cycles
Execution Core	512-entry reorder buffer; 8-wide execute/retire; full bypass network; all instructions have 1-cycle latency except for integer multiply (8-cycles), FP add, FP multiply, FP convert, FP compare, and FP bit operations (each 4-cycle), and FP divide (16 cycles)
On-chip Caches	64KB, 4-way L1 data cache and 2-cycle latency; 1MB, 8-way, unified L2 cache with 8 banks and 6-cycle latency, 1 L2 read port, 1 L2 write port; all caches use LRU replacement and have 64B line size;
Buses and Memory	300-cycle minimum main memory latency; 32 DRAM banks; 32B-wide, split-transaction core-to-memory bus at 4:1 frequency ratio; maximum 64 outstanding misses to main memory; bank conflicts, bandwidth, and queuing delays faithfully modeled

4.2. Support for Predicated Execution on an Out-of-order Processor

In an out-of-order execution processor, a predicated instruction makes register renaming more complicated because it may or may not write into its destination register depending on the value of the predicate [37, 27]. Several solutions have been proposed to handle this problem: (1) converting predicated instructions into C-style conditional expressions [37], (2) breaking predicated instructions into two μops [10], (3) select-μop mechanism [40], (4) predicate prediction [8]. Our baseline processor uses the first mechanism, which has no extra instruction overhead. We also experimented with the select-μop mechanism and found its performance to be similar to the first mechanism (See Section A.3). Note that wish branch can be used with any of these mechanisms. We briefly explain our baseline mechanism below. A full evaluation of the mechanisms used to support predicated execution on an out-of-order processor is out of the scope of this paper.

Converting a predicated instruction into a C-style conditional expression: This mechanism transforms the predicated instruction into another instruction similar to 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 the destination register, which is architecturally a `nop` operation. Hence, regardless

of predicate value, the instruction *always* writes into the destination register, allowing the dependent instructions to be renamed correctly. The advantage of this mechanism is that it has no extra instruction overhead. The disadvantage of this mechanism is that it requires four register sources (old register value, predicate register, and two source inputs). It also increases the length of the dependency chain by adding a `nop` operation when the predicate is `false`. Sprangle and Patt [37] proposed an enhanced mechanism to eliminate the `nop` operation using a statically defined tag.

4.3. Trace Generation and Benchmarks

IA-64 traces were generated with the Pin instrumentation tool [26]. 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 and it is stored using a compressed trace format. We developed an IA-64 to Alpha ISA translator to convert the IA-64 instructions in the trace into μops, which are similar to Alpha ISA instructions. All NOPs are eliminated during μop translation.

All experiments were performed using the SPEC 2000 integer benchmarks. The benchmarks were run with a reduced input set [20].[6] Table 5 shows information about the simulated benchmarks.[7] To reduce the trace generation time and the simulation time, each trace contains about 100M instructions after the initialization phase of the benchmark. We synchronize traces between different binaries using procedure call counts so that we can fairly compare the same portions of the program among different binaries.

Table 5. Simulated benchmarks

Benchmark	Simulated section (based on procedure call count)	Dynamic instructions IA64 instructions / μops	Static branches	Dynamic branches	Mispredicted branches (per 1000 μops)	IPC/μPC
164.gzip	42500 - 765000	94M / 66M	781	10M	8.71	2.25/ 1.56
175.vpr	1 - 1665000	154M / 106M	3525	13M	7.89	2.38/ 1.64
181.mcf	1630000 - 4246000	94M / 66M	1187	14M	6.05	1.52/ 1.07
186.crafty	1 - 1780000	86M / 61M	5231	6M	4.37	1.68 / 1.19
197.parser	2600000 - 5030000	105M / 76M	913	17M	10.32	1.21/ 0.87
253.perlbmk	1 - 700000	195M / 137M	3452	17M	0.07	1.21 / 0.85
254.gap	1 - 450000	93M / 66M	3134	7M	2.47	1.22/ 0.86
255.vortex	395000 - 1685000	93M / 71M	7535	10M	0.77	1.06/ 0.81
256.bzip2	1400000 - 1785000	75M / 52M	634	8M	16.40	1.38/ 0.96
300.twolf	240000 - 780000	119M / 79M	3288	7M	6.5	1.81/ 1.20

4.4. Compilation

All benchmarks were compiled for the IA-64 ISA with the -O2 optimization by the ORC compiler. Software pipelining, speculative load optimization, 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 provides less than 1% performance benefit on the SPEC 2000 integer benchmarks [6] and we removed this optimization to simplify our analysis. Wish branch code generation is also performed with -O2 optimization.[8] To compare wish branches to normal branches and predication, we generated six different binaries, which are described in Table 6. Across all six binaries, forward branches that cannot be predicated remain as normal branches.

[6] Because each experiment uses different binaries, other ways of reducing the simulation time such as SimPoint is not easily applicable in our experiments.

[7] The information shown in Table 5 is collected using the *normal branch binary*, which is described in Table 6. NOPs *are* included in the dynamic IA-64 instruction count, but they are *not* included in the μop count.

[8] Due to problems encountered during compilation and trace generation, gcc and eon benchmarks were excluded. We are currently working on fixing these problems.

Table 6. Description of binaries compiled to assess the performance of different combinations of wish branches

Binary name	Branches that can be predicated with the ORC algorithm [21, 25, 24] ...	Backward branches ...
normal branch binary	remain as normal branches	remain as normal branches
predicated code binary	are predicated	remain as normal branches
wish jump binary	are converted to wish jumps or are predicated (see Section 3.1.1)	remain as normal branches
normal branch and wish loop binary	remain as normal branches	are converted to wish loops or remain as normal branches
predicated code and wish loop binary	are predicated	are converted to wish loops or remain as normal branches
wish jump and wish loop binary	are converted to wish jumps or are predicated (see Section 3.1.1)	are converted to wish loops or remain as normal branches

5. Simulation Results and Analysis

We first evaluate how wish jumps and wish loops perform compared to normal branches and predicated code. Figure 6 shows the normalized number of execution cycles for the six different binaries. Execution time is normalized to the normal branch binary. These results show that the wish jump binary performs as well as or better than the normal branch binary for all benchmarks. The wish jump binary performs 11% better than the predicated code binary for mcf, where the predicated code binary actually degrades performance due to its overhead. The wish jump binary performs slightly worse than the predicated code binary for crafty, vortex, and twolf due to the pipeline flushes caused by wish jump mispredictions and the overhead of extra instructions. Combining normal branches with wish loops performs better than the normal branch binary for vpr, parser, bzip2, and twolf, without degrading performance in any benchmark. Combining predicated code with wish loops improves the performance of the predicated code binary for the same four benchmarks, but does not eliminate the negative performance impact of predicated code in mcf. Using both wish jumps and wish loops removes the negative impact of predication in mcf by utilizing the benefit of wish jumps and also further increases performance by utilizing the benefit of wish loops, with a maximum performance improvement of 19% over the normal branch binary in vpr and twolf. On average, utilizing both wish jumps and wish loops obtains the best performance across the six binaries, with an average improvement of 7.2% over the normal branch binary and 2.4% over the predicated code binary.

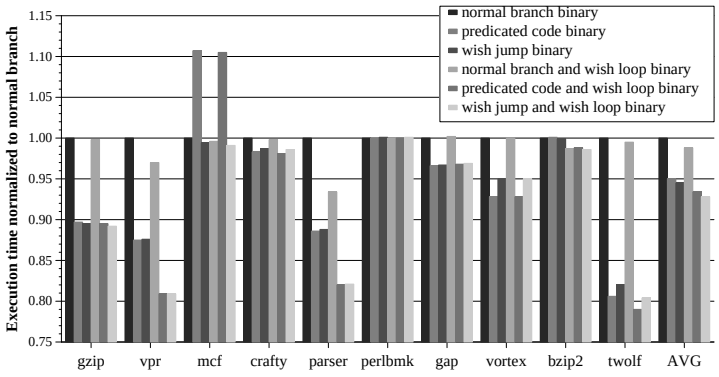


Figure 6. Relative execution cycles normalized to normal branch binaries

Figure 7 shows the distribution of dynamic mispredicted branches in the normal branch binary based on how they are treated in the other binaries. The bar labeled *Predicated* shows the mispredicted branches that are predicated in both the predicated code binary and the binaries containing wish jumps. *Wish Jump* shows the branches that are converted to wish jumps in the binaries containing wish jumps. *Wish Loop* shows the branches that become wish loops in the binaries containing wish loops. *Normal Branch* shows the branches that remain as normal conditional branches in all six binaries. Figure 7 provides insight into why the wish jump binary and the predicated code binary perform better than the normal branch binary: in gzip, vpr, parser, vortex, and twolf, more than 20% of the mispredicted branches in the normal branch

binary are eliminated using predication in the wish jump binary and the predicated code binary. Figure 7 also shows that few mispredicted branches are converted to wish jumps, because many branches either become predicated code or cannot be converted to predicated code. We discuss how a better compiler can convert more mispredicted branches into wish jumps in Section 6.1.

Figure 7 also provides insight into why using wish loops increases performance in vpr, parser, bzip2, and twolf. Approximately 15%, 42%, 10%, and 15% of the mispredicted branches in vpr, parser, bzip2, and twolf, respectively are converted to wish loops. Section 5.2 examines the behavior of mispredicted wish loop instructions in detail.

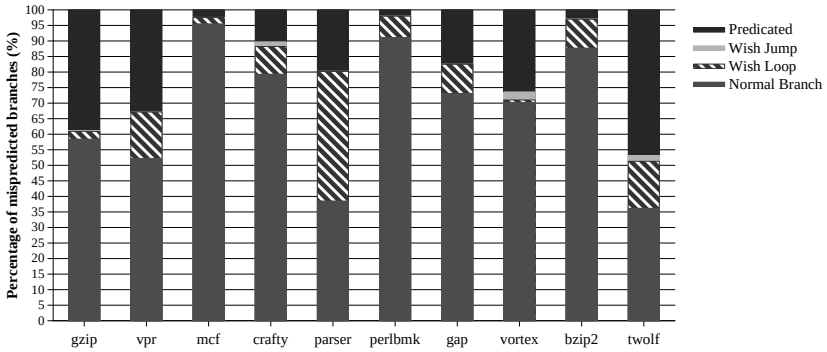


Figure 7. Mispredicted branch distribution in normal branch binary

5.1. Benefits of Wish Jumps

Figure 8 shows the dynamic distribution of the four cases that were analyzed in Table 1. Remember that a wish jump performs better than predicated execution in the PTAT case. Figure 8 shows that 90% of the wish jumps in mcf fall into the PTAT category. We find that, in mcf, many of the branches that are predicated in the predicated code binary are actually correctly predicted in the normal branch binary. Using wish jumps instead of predication for these branches eliminates the instruction overhead of predication for the PTAT correct predictions, which results in significant performance improvement over predicated code. Figure 8 also shows that the case in which wish jump performs worse than predicated execution (PTAN) does not occur frequently. The performance of wish jumps and predicated execution are similar in the PNAN and PNAT cases, but a wish jump has an extra instruction overhead. We find that this overhead is the cause of the performance loss of the wish jump binary compared to the predicated code binary in twolf and vortex. Figure 8 shows that more than 50% of the wish jumps are actually not taken (PNAN and PTAN) for vpr, parser, perlbmk, gap, vortex, and twolf. As wish jumps are more beneficial when they are taken, the performance benefit of wish jumps in these benchmarks can increase with smarter profile-based code reorganization in which the compiler optimizes the wish jump code such that the taken case is more frequent than the not-taken case.

5.2. Benefits of Wish Loops

Figure 9 shows the dynamic distribution of the four cases that were analyzed in Table 3. Figure 9 shows that approximately 50% of the mispredicted wish loops fall into the late-exit category, which provides performance improvement in vpr, parser, bzip2, and twolf. Since the early-exit case constitutes a significant percentage of the mispredicted wish loops, a loop predictor specialized for converting the harmful early-exit cases to the beneficial late-exit cases has the potential to increase the performance benefit of wish loops.

Because the predication of the wish loop body causes extra execution delay and because wish loop conversion adds an extra instruction to the loop header, performance reduction is possible if the compiler does not carefully convert branches

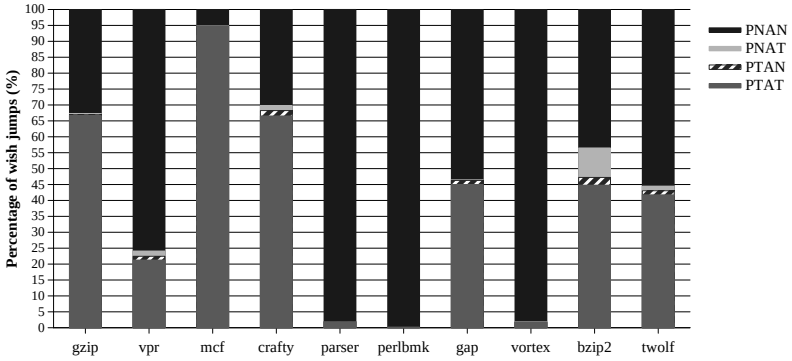


Figure 8. Dynamic distribution of wish jumps (based on wish jump binary)

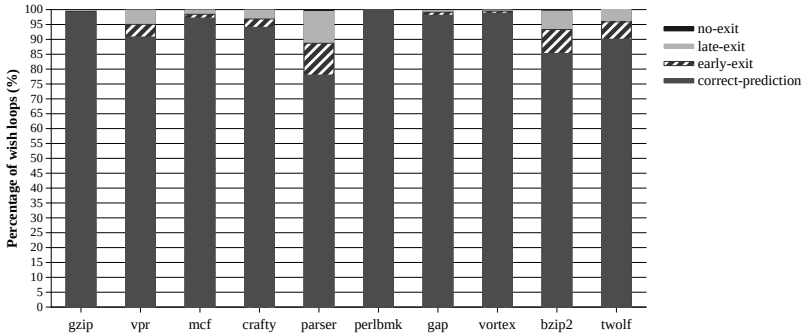


Figure 9. Dynamic distribution of wish loops (based on *normal branch and wish loop* binary)

to wish loops. Our baseline compiler algorithm converts a backward branch to a wish loop when the size of the loop body is less than or equal to N instructions where N=5. Figure 10 shows the execution time when N is varied from 10 to 50. When larger loops are converted to wish loops, a performance loss is observed across all benchmarks, especially in mcf and perlbmk. To identify the cause of the overhead in wish loops, two ideal experiments are performed. First, we ideally remove the cost of the extra instruction inserted into the loop header in our simulator (NOINIT - second bar from the right in Figure 10). Second, in addition, we ideally eliminate the execution delay caused by the predication of the loop body in our simulator (NOINIT + NODELAY - rightmost bar). Eliminating the extra execution delay removes the performance degradation observed in all benchmarks when larger loops are converted to wish loops. In contrast, eliminating the extra instruction does not affect performance, implying that the major overhead of the wish loop is the execution delay it causes due to predication. Figure 10 also shows that our simple compiler heuristic works well in that it achieves almost the same performance achieved when the overhead due to wish loops is eliminated.

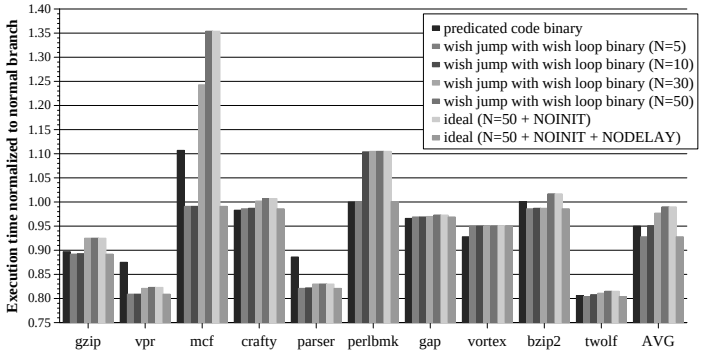


Figure 10. The overhead due to wish loops

5.3. Hardware Optimizations for Wish Branches

5.3.1. Predicate dependency elimination If a wish jump is predicted taken, the predicate value can be predicted to be *true*. If the wish jump is correctly predicted, the instructions on the taken path do not need to wait until the predicate value is computed before they are executed. If the wish jump is mispredicted, the processor flushes the pipeline so the instructions after the wish jump will be re-executed with the correct predicate value. The prediction mechanism is as follows: When a wish jump is predicted taken, the predicate register number of the wish jump 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. The special buffer is reset if there is a branch misprediction or if an instruction that writes to the same predicate register is decoded. For example, in Figure 1c, when the wish jump is predicted to be taken, the predicate register number p1 is stored in the special buffer. The following instruction *(p1) mov, b0* has the same predicate register number p1, so it can be executed before the instruction that generates the *p1* value (*p1 = cond*) is executed. Figure 11 shows that this optimization results in about 1% improvement for gzip and crafty over the wish jump binary.

5.3.2. Confidence estimator Wish jumps can be better utilized with a confidence mechanism. When the prediction made for a wish jump has low confidence, the wish jump is predicted to be not taken. This reduces the occurrence of the PTAN case, which requires a pipeline flush, and increases the occurrence of the PNAT or PNAN cases. Figure 11 shows the performance of wish jumps with a perfect branch confidence mechanism, where every mispredicted wish jump is predicted to be low-confidence. Since the PTAN case does not occur frequently as shown in Figure 8, the performance improvement is negligible even with a perfect confidence predictor.

5.3.3. Wish joins As explained in Section 2.2, wish joins should be used with a confidence estimator. To demonstrate the potential of wish joins, we use a perfect confidence predictor. Figure 11 shows that, even with a perfect confidence predictor, wish joins do not show significant additional performance benefit in our current wish jump binaries. We found that the number of wish join instructions inserted is negligible due to the reasons explained in Section 3.2. Hence, the performance impact of the wish joins is negligible.

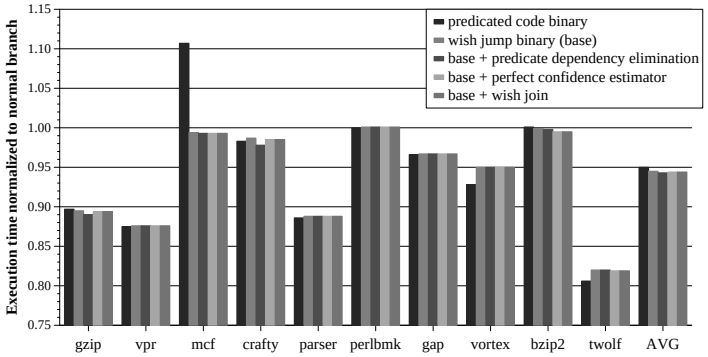


Figure 11. Relative execution cycles with hardware optimizations for wish branches

5.4. Hardware Optimizations for Wish Loops

A specialized wish loop predictor can be designed to overestimate the iteration count of a loop to make the late-exit case more common than the early-exit case when a wish loop is hard to predict. This predictor would take advantage

of the benefits provided by the wish loop without requiring the exact prediction of the iteration count of a loop. We propose a specialized predictor design based on the previously-proposed loop termination predictor [34]. In contrast to the previous proposal, the wish loop predictor does not try to predict the *exact* number of iterations of a loop. It predicts that the wish loop will iterate the *maximum* number of times it iterated in the past. For example, if a wish loop iterated 3, 6, 4, 4, 2, 1, and 3 times in the last seven executions of the loop, the wish loop predictor predicts that the loop will iterate 6 times in the next execution.

The wish loop predictor, shown in Figure 12, has three counters: SpecIter (number of speculatively fetched iterations), NonSpec (number of retired iterations), and ConfCount (a counter used to determine the confidence of the loop prediction). The maximum number of retired loop iterations encountered for each loop is stored in the MaxIter field. SpecIter is incremented by one when the wish loop branch is predicted to be taken in the front end. NonSpec is incremented by one when the wish loop branch retires and is found to be taken. When the wish loop retires and is found to be not-taken, the NonSpec value is compared with the Max value. If the NonSpec value is less than or equal to the Max value, ConfCount is incremented. If the NonSpec value is greater than the Max value, Max value is set to the NonSpec value and both ConfCount and Conf are reset to zero. NonSpec and SpecIter are reset to zero when a wish loop branch retires and is found to be not-taken. Once the ConfCount value reaches a threshold N (N=5 in our simulations), the Conf bit is set. The output of the loop predictor is valid only if the Conf bit is set.

A wish loop branch is predicted taken if the SpecIter value is less than the MaxIter value. When the SpecIter counter reaches the MaxIter value, the wish loop branch is predicted not-taken. This prediction is used only if the confidence estimation for the main branch predictor's prediction is low confidence. If the confidence estimation for that prediction is high confidence, the loop predictor's result is discarded and the branch predictor's prediction is used.

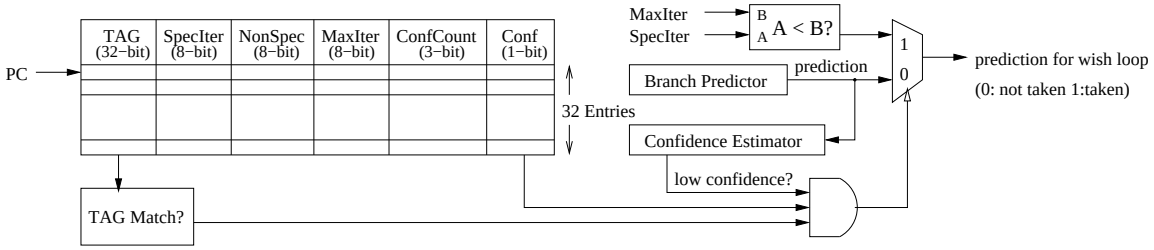


Figure 12. Wish loop prediction mechanism

Figure 13 shows the execution time when the wish loop predictor is utilized along with a realistic confidence estimator [16]. Compared to the execution time of the *wish jump and wish loop binary* that does not utilize the wish loop predictor, a 4% performance improvement is observed for parser and a 1-2% performance improvement is observed for vpr, bzip2, and twolf. Similar improvements are seen with the loop predictor on the *predicated code and wish loop binary*. The reason is that the wish loop predictor migrates early-exit cases to late-exit cases. For example, in parser, without the proposed loop predictor, 11% of all mispredicted branches fall into the late-exit case and 11% fall into the early-exit case (Figure 9). When the wish loop predictor is used, 14% of all mispredicted branches fall into the late-exit case and 6% fall into the early-exit case. With the wish loop predictor, utilizing wish jumps and wish loops results in an average improvement of 7.8% (maximum 21% in vpr, parser, and twolf) over the normal branch binary and 3% (maximum 11% in mcf and parser) over the predicated code binary.

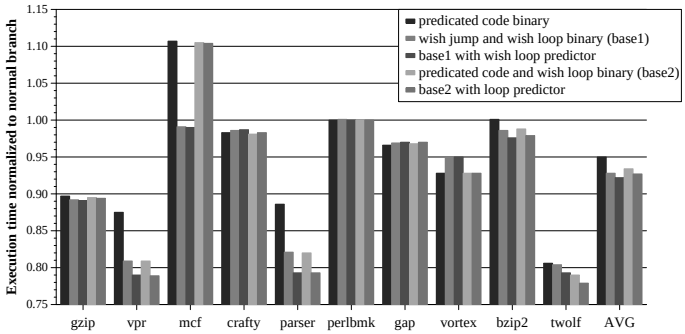


Figure 13. Relative execution cycles with the wish loop predictor

6. Discussion

This section examines the shortcomings of our implementation, provides insights into how wish branches can perform better by reducing some of these shortcomings, and proposes areas for future study.

6.1. Compiler Effects on Wish Jump Effectiveness

As discussed in Section 3, a branch instruction can be converted to a wish jump only if it can be predicated. Since our base compiler uses a region-based compilation algorithm, all if-conversion is performed within a region boundary [21, 25]. How a region is constructed is a critical factor that determines how many branches can be predicated. A different region formation algorithm can convert more branches to predicated code, which also increases the opportunities for converting branches to wish jumps.

The region formation algorithm built into the base compiler is also not optimized for building large predicated code blocks. In predicated execution, large predicated blocks can hurt performance because they result in the execution of a large number of useless instructions. As we have shown, wish jumps can reduce this problem associated with large predicated blocks. If a compiler produces larger predicated code blocks by using a different region formation algorithm or different configuration values in the region formation algorithm, this will increase the wish fall-through block size. A larger wish fall-through block will provide more opportunities for a wish branch to improve the performance of predicated execution and thus may make predication of larger code blocks more viable using the wish branch mechanism. Hence, future research should evaluate the impact of region formation algorithms on the effectiveness of wish jumps and predication.

6.2. Wish Jumps vs. Predicated Code, Revisited

One advantage of predication is that it gives the compiler more freedom and scope in code optimization by combining multiple basic blocks into a single basic block [14, 23]. If we use wish jumps instead of predication alone, this code optimization benefit is reduced, because a wish jump breaks the code into multiple basic blocks instead of using a single large basic block. Therefore, predication can have additional benefit over wish jumps, since it allows for the generation of more optimized code. Because we evaluate both wish jumps and predicated execution on an aggressive out-of-order, superscalar processor model, we found that the additional code optimization benefit present in the predicated code does not significantly affect performance. For other processor models (e.g. VLIW, in-order), the additional benefit due to predication can be more significant. Therefore, the tradeoffs between predication and wish jumps in other processor models needs to take into account the code optimization effects.

Another advantage of predication is that it reduces the pressure on the branch predictor by eliminating branches. This

may reduce the branch mispredictions for the remaining branches in the code. As wish jumps add more branches to the code, this advantage provided by predication may be reduced. On the other hand, the addition of wish jumps to the code can actually help increase the prediction accuracy because the prediction of wish jumps adds more information to the global branch history register that is used to make branch predictions. Previous research has shown that information on the direction of the predicated branches does increase the branch prediction accuracy [3, 35]. We find that these conflicting effects balance each other and the additional branches present in the wish jump code do not significantly affect the branch predictor accuracy.

6.3. Global Branch History Register Updates

In current microprocessors, the global branch history register (GHR) is updated speculatively in the fetch stage and is recovered using the correct branch direction during branch misprediction recovery [17]. As such, branch predictions made on the mispredicted path do not corrupt the history used to predict branches that are on the correct program path. In contrast, a processor using wish branches is sometimes unable to recover the GHR on a wish jump or wish loop misprediction. This happens in the PNAT case for the wish jump and in the late-exit case for the wish loop. For both of these cases, no misprediction recovery is initiated even though the wish jump/loop is mispredicted. Hence, the bit corresponding to the wish jump/loop in the GHR cannot be corrected. This results in the prediction of future branches with incorrect GHR values, which may result in an increase in the number of branch mispredictions. This problem can affect wish loops more significantly, because multiple wish loops can update the GHR speculatively and they can all be on the mispredicted path. Therefore, multiple bits in the GHR may be incorrect in the late-exit case. We found that eliminating the negative effects of this problem has the potential to improve the performance provided by the wish branches. Hence, part of our future research is on techniques to *fix* the GHR upon wish branch mispredictions.

7. Related Work

7.1. Related Research on Predicated Execution

Predicated execution was first implemented in the Cray-1 computer system as *mask vectors* [33]. Allen et al. [1] proposed the predication of instructions using *if conversion* to enable automatic vectorization in the presence of complex control flow. Hsu and Davidson proposed the use of predicated execution for scalar instructions, which they called *guarded execution*, to reduce the penalty of conditional branches in deeply-pipelined processors [14]. Hsu and Davidson also described how predicated execution enables compiler-based code scheduling optimizations.

Several papers examined the impact of predicated execution on branch prediction and instruction-level parallelism. Pnevmatikatos and Sohi [27] 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. [22], Tyson [38], and Chang et al. [4] showed that predicated execution can eliminate a significant number of branch mispredictions and can therefore reduce the program execution time.

Choi et al. [6] 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 2000 integer benchmark suite, it results in only a 2% improvement in average execution time. For some benchmarks, a performance loss is observed with predicated execution. This performance loss and small performance gain is due to the *overhead* of predication (the extra useless instructions executed). Our paper aims to reduce the overhead of predication by using the wish branches to dynamically eliminate the useless predicated instructions.

Chuang and Calder [8] proposed a hardware mechanism to predict the predicate values in order to overcome the multiple-definition problem [37, 40] in an out-of-order processor that implements predicated execution. Although they do not mention it, their mechanism can reduce the extra instruction overhead of predicated execution, if the predicate value is easy to predict at run time. However, if the predicate value is hard to predict, incorrect predicate predictions will result in partial pipeline flushes, which reduce the benefit provided by predicated execution. In contrast to predicate prediction, which is performed for every predicate, a wish branch is inserted by the compiler only for a subset of the predicated branches. With wish branches, the hardware has the freedom to dynamically decide whether to use conditional branching or predicated execution, whereas with predicate prediction every predicate is predicted, just like a normal conditional branch.

Klauser et al. [18] proposed *dynamic hammock predication (DHP)*, which is a hardware mechanism that dynamically predicates hammock branches. This mechanism supports predication on processors with little or no ISA predication support. Like wish branches, DHP provides the hardware with the ability to dynamically decide whether or not to predicate a hammock branch. In contrast to wish branches, DHP is a hardware-based mechanism and adds significant hardware complexity to the pipeline, especially to the renaming logic. In this paper, we assume predication support exists in the ISA and try to reduce the negative effects of predication by using combined software/hardware mechanisms, without significantly increasing the hardware complexity.

7.2. Related Research on Control Flow Independence

Several hardware mechanisms have been proposed to exploit control flow independence [30] by reducing the branch misprediction penalty or improving parallelism [29, 30, 31, 7, 5, 12]. Similar to the wish branch's objective, these techniques aim to avoid flushing the processor pipeline when the processor is known to be on the correct control flow path 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. Hardware is required for the following:

1. Detection of the reconvergent (control-flow independent) point in the instruction stream: While some mechanisms use software to detect the reconvergent point [30, 31], most proposed mechanisms use hardware-based heuristics and predictors [29, 7, 5, 12, 9]. The hardware used to detect/predict the reconvergent point adds more complexity to the processor pipeline. In contrast, a wish branch *exactly* specifies the reconvergent point, because the compiler that generates the wish branch knows *exactly* where the reconvergent point is in the instruction stream. Hence, there is no need for extra hardware.
2. Removal of wrong-path instructions, formation of correct data dependences for control-independent instructions, and selective re-scheduling and re-execution of instructions: Proposed mechanisms to exploit control flow independence [29, 30, 31, 7, 5, 12] require fairly complicated hardware structures to accomplish these tasks [30]. In contrast, as wish branches make use of predication to exploit control-flow independence, there is no need to provide extra hardware other than what is in place to support predicated execution. Instructions that are on the wrong-path will become NOPs because they are predicated, and the control-independent instructions on the correct path already have the correct data dependences, because the compiler correctly identifies their dependences while generating predicated code, which eliminates the need for re-scheduling and re-execution. In summary, wish branches, with their use of predication, eliminate most of the complex hardware support required to exploit control-flow independence purely in hardware.

Instruction reuse buffers [36] and register integration [32] are two mechanisms that reuse the execution results of the control and data independent instructions that are on the wrong path. Unlike wish branches, in these two approaches, the pipeline needs to be flushed on a branch misprediction.

7.3. Related Research 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 [28] 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 [13], disjoint eager execution [39], and the PolyPath architecture [19] refined eager execution to reduce its implementation cost. These mechanisms are at a level similar to the wish branch mechanism because they fetch and execute 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 (i.e., the not-taken path in wish jump code) by predicating the code. No hardware support is needed to fetch from multiple paths. The hardware simply needs to decide whether to fetch instructions from the taken path or from the not-taken path. In multipath execution, extra hardware resources are needed to fetch and execute from multiple control-flow paths.

8. Conclusion

This paper proposes a new control-flow instruction called a *wish branch* to reduce the branch misprediction penalty. We introduce three types of wish branches (wish jump, wish join, and wish loop) and provide insights into how and why they can perform better than normal branch prediction and predicated execution. The compiler algorithms to generate wish jumps, wish joins, and wish loop branches are described. We demonstrate the performance improvement of wish branches by implementing the new instructions in a state-of-the-art compiler.

Using wish jumps and wish loops together decreases the execution time of the SPEC 2000 integer benchmarks by 7.2% (up to 19%) compared to traditional conditional branches and by 2.4% (up to 11%) compared to predicated execution, without requiring complex hardware support. To improve the benefit of wish loops, we propose a specialized wish loop predictor. With this predictor, the execution time is reduced by 7.8% (up to 21%) compared to traditional conditional branches and 3% (up to 11%) compared to predicated execution. We also describe several hardware optimizations which can improve the performance benefit of the wish jumps. Since few mispredicted branches are converted to wish jumps, these hardware optimization techniques do not show significant performance benefits. These techniques will be more useful with better compilation techniques for generating wish jump code, which we intend to investigate as part of our future research.

9. References

[1] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *10th ACM Symposium on Principles of Programming Languages*, pages 177–189, 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] D. I. August, D. A. Connors, J. C. Gyllenhaal, and W. mei W. Hwu. Architectural support for compiler-synthesized dynamic branch prediction strategies: Rationale and initial results. In *Proceedings of the Third IEEE International Symposium on High Performance Computer Architecture*, 1997.

[4] 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 *Proceedings of the 1995 ACM/IEEE Conference on Parallel Architectures and Compilation Techniques*, 1995.

[5] C.-Y. Cher and T. N. Vijaykumar. Skipper: a microarchitecture for exploiting control-flow independence. In *Proceedings of the 34th ACM/IEEE International Symposium on Microarchitecture*, pages 4–15, 2001.

[6] 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 *Proceedings of the 34th ACM/IEEE International Symposium on Microarchitecture*, 2001.

[7] Y. Chou, J. Fung, and J. P. Shen. Reducing branch misprediction penalties via dynamic control independence detection. In *Proceedings of the 13th International Conference on Supercomputing*, pages 109–118, 1999.

[8] W. Chuang and B. Calder. Predicate prediction for efficient out-of-order execution. In *Proceedings of the 17th International Conference on Supercomputing*, pages 183–192, 2003.

[9] J. D. Collins, D. M. Tullsen, and H. Wang. Control flow optimization via dynamic reconvergence prediction. In *Proceedings of the 37th ACM/IEEE International Symposium on Microarchitecture*, 2004.

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

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

[12] A. Gandhi, H. Akkary, and S. Srinivasan. Reducing branch misprediction penalty via selective recovery. In *Tenth International Symposium on High Performance Computer Architecture*, 2004.

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

[14] P. Hsu and E. Davidson. Highly concurrent scalar processing. In *Proceedings of the 13th International Symposium on Computer Architecture*, 1986.

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

[16] E. Jacobsen, E. Rotenberg, and J. E. Smith. Assigning confidence to conditional branch predictions. In *Proceedings of the 29th ACM/IEEE International Symposium on Microarchitecture*, pages 142–152, 1996.

[17] S. Jourdan, T.-H. Hsing, J. Stark, and Y. N. Patt. The effects of mispredicted-path execution on branch prediction structures. In *Proceedings of the 1996 ACM/IEEE Conference on Parallel Architectures and Compilation Techniques*, pages 58–67, 1996.

[18] A. Klauser, T. Austin, D. Grunwald, and B. Calder. Dynamic hammock predication for non-predicated instruction set architectures. In *Proceedings of the 1998 ACM/IEEE Conference on Parallel Architectures and Compilation Techniques*, 1998.

[19] A. Klauser, A. Paithankar, and D. Grunwald. Selective eager execution on the polypath architecture. In *Proceedings of the 25th International Symposium on Computer Architecture*, 1998.

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

[21] Y. Liu, Z. Zhang, R. Qiao, and R. Ju. A region-based compilation infrastructure. In *Proc. of the 7th Workshop on Interaction between Compilers and Computer Architecture*, 2003.

[22] 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 ACM/IEEE International Symposium on Microarchitecture*, pages 217–227, 1994.

[23] 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 ACM/IEEE International Symposium on Microarchitecture*, pages 45–54, 1992.

[24] S. Mantripragada and A. Nicolau. Using profiling to reduce branch misprediction costs on a dynamically scheduled processor. In *Proceedings of the 2000 International Conference on Supercomputing*, pages 206–214, 2000.

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

[26] Pin. *A Binary Instrumentation Tool*. http://rogue.colorado.edu/Pin/index.php.

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

[28] 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.

[29] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. E. Smith. Trace processors. In *Proceedings of the 30th ACM/IEEE International Symposium on Microarchitecture*, 1997.

[30] E. Rotenberg, Q. Jacobson, and J. E. Smith. A study of control independence in superscalar processors. In *Proceedings of the Fifth IEEE International Symposium on High Performance Computer Architecture*, 1999.

[31] E. Rotenberg and J. Smith. Control independence in trace processors. In *Proceedings of the 32nd ACM/IEEE*

International Symposium on Microarchitecture, 1999.

[32] A. Roth and G. S. Sohi. Register integration: A simple and efficient implementation of squash reuse. In *Proceedings of the 33rd ACM/IEEE International Symposium on Microarchitecture*, 2000.

[33] R. M. Russell. The CRAY-1 computer system. *Communications of the ACM*, 21(1):63–72, Jan. 1978.

[34] T. Sherwood and B. Calder. Loop termination prediction. In *Proceedings of the Third International Symposium on High Performance Computing*, 2000.

[35] B. Simon, B. Calder, and J. Ferrante. Incorporating predicate information into branch predictors. In *Proceedings of the Ninth IEEE International Symposium on High Performance Computer Architecture*, 2003.

[36] A. Sodani and G. S. Sohi. Dynamic instruction reuse. In *Proceedings of the 24th International Symposium on Computer Architecture*, 1997.

[37] E. Sprangle and Y. Patt. Facilitating superscalar processing via a combined static/dynamic register renaming scheme. In *Proceedings of the 27th ACM/IEEE International Symposium on Microarchitecture*, pages 143–147, 1994.

[38] G. S. Tyson. The effects of predication on branch prediction. In *Proceedings of the 27th ACM/IEEE International Symposium on Microarchitecture*, pages 196–206, 1994.

[39] A. Uht and V. Sindagi. Disjoint eager execution: An optimal form of speculative execution. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 313–325, 1995.

[40] 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 *Proceedings of the Seventh IEEE International Symposium on High Performance Computer Architecture*, 2001.

A. Sensitivity of the Performance of Wish Branches to Microarchitectural Parameters

A.1. Effect of Instruction Window Size and Pipeline Depth

We analyze the benefits of wish branches on machines with smaller instruction windows and shorter pipelines. Figure 14 shows the normalized execution time of the five binaries on three different machines with 128, 256, and 512-entry instruction windows. The data is averaged over all the ten benchmarks examined. 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. We can see that 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 branches are also more effective on larger windows, because it is more likely to have useful instructions in the pipeline of a machine with a larger window when a mispredicted branch is resolved. Hence, the potential of exploiting control-flow independence is higher on a larger instruction window.

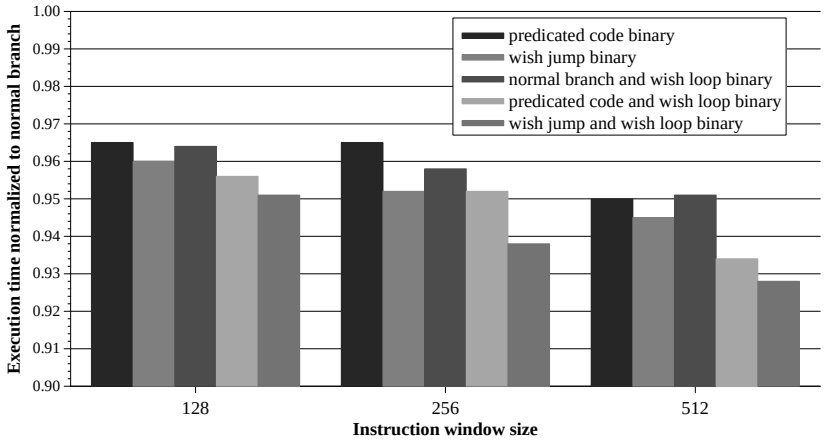


Figure 14. Sensitivity of wish branch performance to instruction window size.

Figure 15 shows the normalized execution time of the five binaries on three different machines with 10, 20, and 30 pipeline stages. The benefits of wish branches increases as pipeline depth increases. Note that the binaries with wish jumps and wish loops always outperform the other binaries for all pipeline depths and instruction window sizes.

A.2. Wish Branches on In-order Processors

We also evaluate the benefits of wish branches in an in-order machine. The processor we evaluate has a 30-cycle minimum branch misprediction penalty. Since branch mispredictions are less costly on an in-order machine predicated code binaries do not show performance benefits as large as they do on out-of-order machines. Even so, wish branches still reduce most of the negative effects of predicated code and keep the benefits of predicated code if the predicated code provides a performance benefit. Wish jumps and wish loops together improve the performance of the in-order processor by 1.8% compared to traditional conditional branches and by 1.1% compared to predicated execution. Notice that using wish loops sometimes significantly reduces performance on an in-order processor, especially in vpr. This is due to the

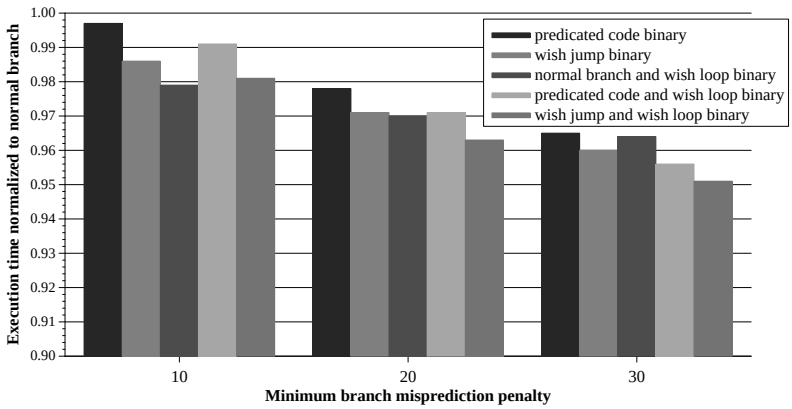


Figure 15. Pipeline depth effect

execution delay caused by the predication of the loop body. The cost of execution delay due to wish loops is much higher in an in-order machine because an in-order machine cannot tolerate the execution delay unlike an out-of-order machine.

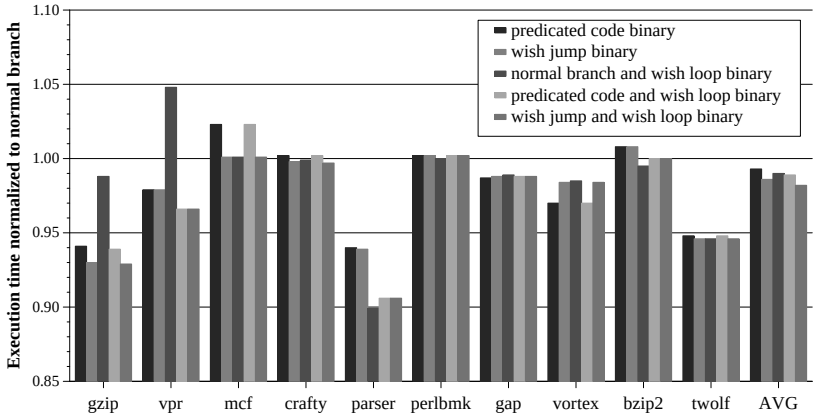


Figure 16. Relative execution cycles normalized to normal branch binaries in an in-order machine

A.3. Effect of the Select-μop Mechanism to Support Predicated Execution

Our baseline out-of-order processor uses C-style conditional expressions to handle predicated instructions as discussed in section 4.2. We also implemented the select-μop mechanism proposed by Wang et. al [40] 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 higher on a processor that supports predicated instructions by converting predicated instructions into C-style conditional expressions.

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 17 shows the normalized execution time of the five different binaries on a processor that supports predicated execution using the select-μop mechanism. All execution times are normalized to the execution time of the binary that contains only conditional branches. Hence, the results in Figure 17 are directly comparable to results in Figure 6 which assumes the base processor that uses C-style conditional expressions to support predicated execution. Predicated code binary provides 2.2% performance improvement on the processor that uses the select-μop mechanism compared to 5% on the baseline processor. Wish jump and wish loop binary improves the performance of the processor that uses the select-μop mechanism by 5.6% compared to normal branch binary and 3.5% compared to predicated code binary.

Hence, on the processor that uses the select-μop mechanism, the overall performance improvement of wish branches over conditional branch prediction (5.6%) is smaller than it is on the processor that uses C-style conditional expressions (7.2%). This is due to the higher overhead of the select-μop mechanism to support predicated instructions. On the other hand, the overall performance improvement of wish branches over predicated execution (3.5%) is larger than it is on the processor that uses C-style conditional expressions (2.4%). Hence, the performance improvement provided by wish branches is larger when predicated execution has higher overhead.

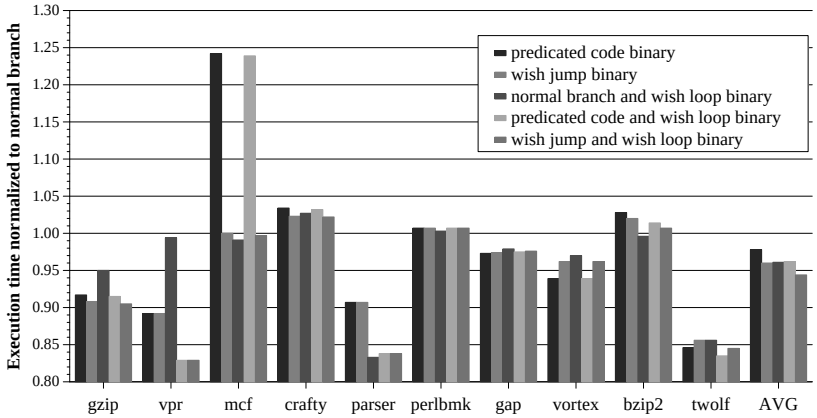


Figure 17. Relative execution cycles normalized to normal branch binaries in select-μop mechanism