Wish Branches: Enabling Adaptive and Aggressive 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

Wish branches, a new class of control-flow instructions, allow the hardware to dynamically decide whether or not to use predicated execution for a dynamic branch instruction. 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. Wish loops, one class of wish branches, utilize predication to reduce the misprediction penalty for hard-to-predict backward (loop) branches.

1. Introduction

Predicated execution has been used to avoid the performance loss due to hard-to-predict branches. This approach eliminates a hard-to-predict branch from the program by converting the control dependency of the branch into a data dependency [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.

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.

2. Sidebar: Background on Microarchitectural Support for Predicated Execution in Out-of-order Execution Processors

Although predicated execution has been implemented in in-order processors [2, 3], as earlier researchers have suggested, the technique can be used in out-of-order processors as well [4, 5]. 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. We briefly provide background information on the microarchitecture support needed to use predicated execution in an out-of-order processor.

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 [4]. Several solutions have been proposed to handle this problem: converting predicated instructions into C-style conditional expressions [4], breaking predicated instructions into two μops [6], the select-μop mechanism [5], and predicate prediction [7]. We briefly describe our baseline mechanism, C-style conditional expressions.

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).

3. The Overhead of Predicated Execution

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. In our previous paper [8], we analyzed the performance impact of these two sources of overhead on an out-of-order processor model that implements predicated execution. We showed that these two sources of overhead, especially the execution delay due to predicated instructions, significantly reduce the performance benefits of predicated execution. When the overhead of predicated execution is faithfully modeled, the predicated binaries do not improve the average execution time of a set of SPEC CPU2000 integer benchmarks. In contrast, if all overhead is ideally eliminated, the predicated binaries would provide 16.4% improvement in average execution time.

In addition, one of the limitations of predicated execution is 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, 9].

In this article, we propose wish branches (1) to dynamically reduce the sources of overhead in predicated execution and (2) to make predicated execution applicable to backward branches, thereby increasing the viability and effectiveness of predicated execution in high-performance, out-of-order execution processors.

4. Wish Branches

4.1. Wish Jumps and Wish Joins

Figure 1 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 1c), but they are not predicated in the normal branch code (Figure 1a). 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 1b) 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 1c) 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.

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.

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.

4.2. Wish Loops

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

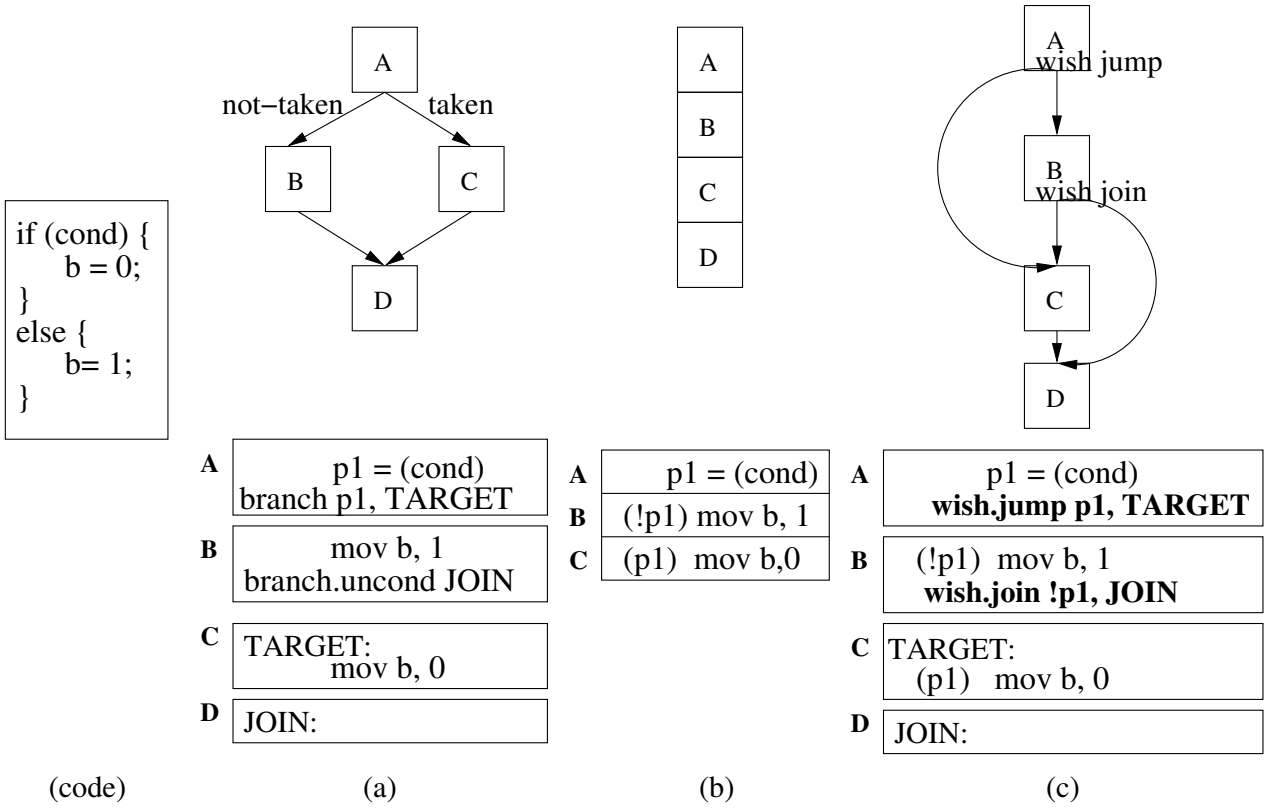


Figure 1. Source code and the corresponding control flow graphs and assembly code for (a) normal branch code (b) predicated code (c) wish jump/join code.

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 2a) and the wish loop code (Figure 2b) is that in the wish loop code 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 2. Similarly, a `while` loop or a `for` loop can also utilize a wish loop instruction.

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 2b) 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

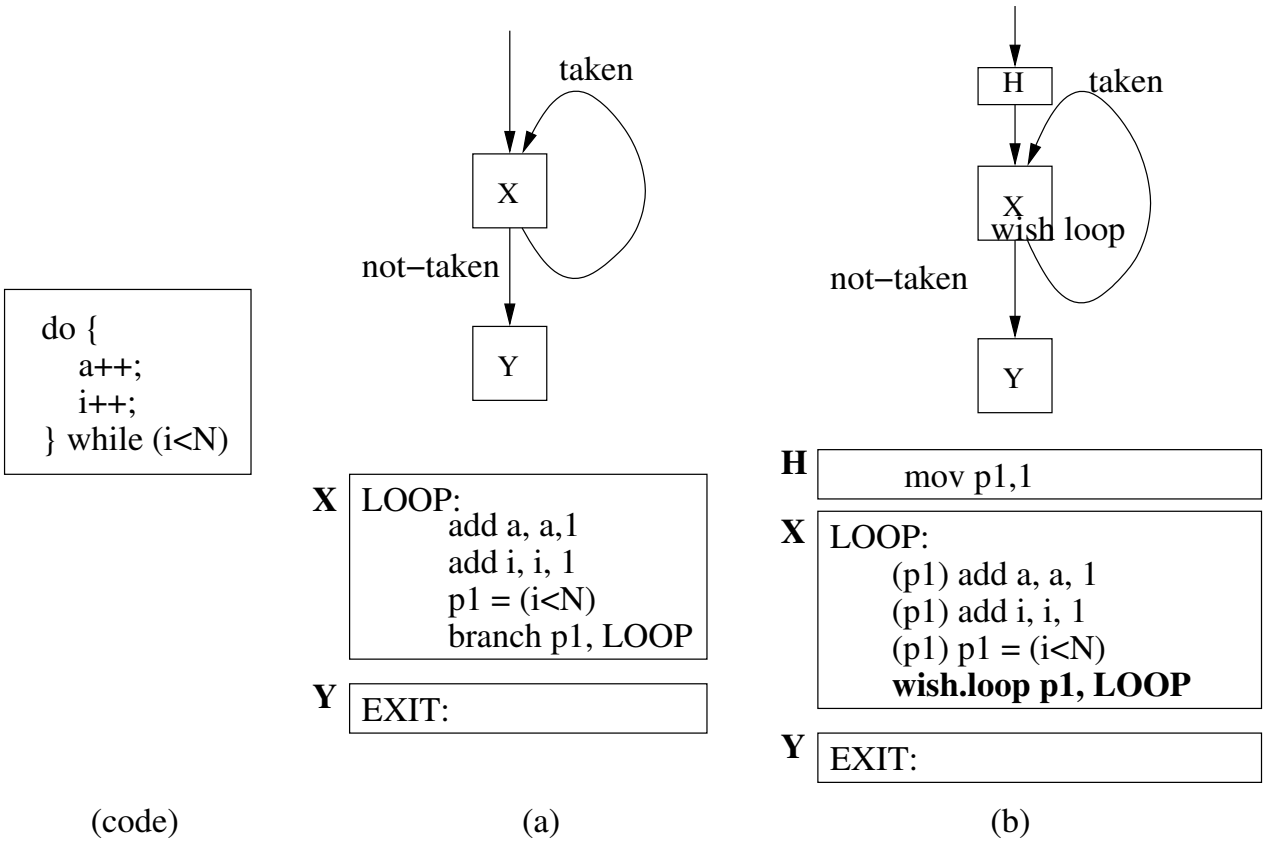


Figure 2. `do-while` **loop source code and the corresponding control flow graphs and assembly code for (a) normal backward branch code (b) wish loop code.**

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.

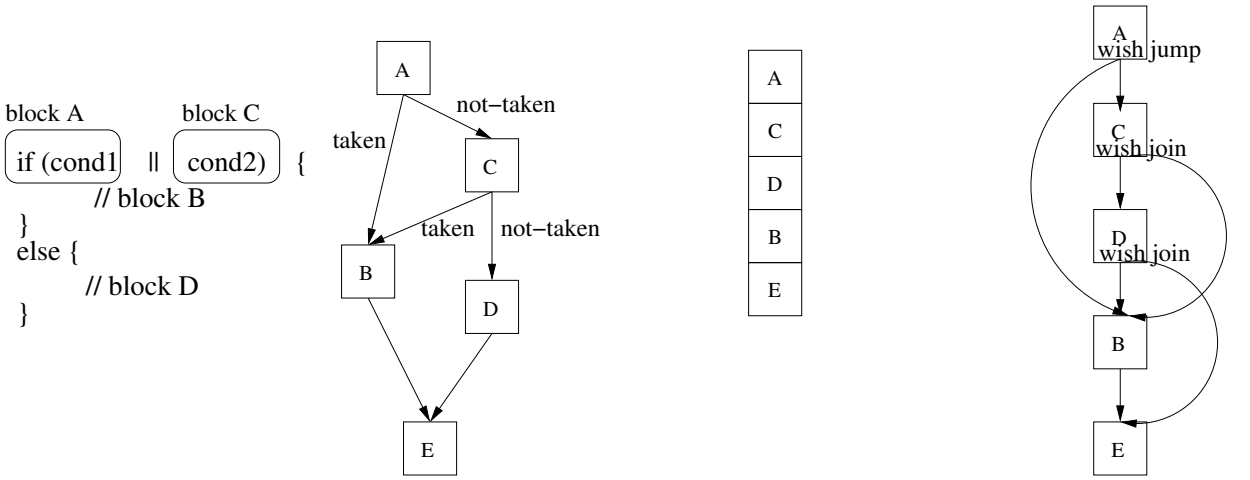
In the late-exit case, the fall-through block Y has been fetched before the predicate for the first extra block X_4 has been resolved. Therefore it is more efficient to simply allow X_4 and subsequent extra block X_5 to flow through the data path as NOPs (with predicate value p1 = FALSE) than to flush 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.

4.3. 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 3 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, 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,



(code) (a) normal branch code (b) predicated code (c) wish branch code

Figure 3. Control flow graph examples with wish branches.

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. Otherwise, the current wish join is predicted using the branch predictor.

4.4. Support for Wish Branches

Since wish branches are an ISA construct, they require support from the ISA, the compiler, and the hardware.

4.4.1. ISA Support We assume that the baseline ISA supports predicated execution. Wish branches can be implemented in the existing branch instruction format using the hint bit fields. Two hint bits are necessary to distinguish between a normal branch, a wish jump, a wish join, and a wish loop.

4.4.2. Compiler Support The compiler needs to support the wish branch code generation algorithm. The algorithm decides which branches are predicated, which are converted to wish branches, and which stay as normal branches based on estimated branch misprediction rates, compile-time heuristics, and information about branch behavior [8].

4.4.3. Hardware Support An accurate confidence estimator [10] is essential to maximize the benefits of wish branches. In addition, wish branches require hardware support in the processor front-end and the branch misprediction detection/recovery module. Detailed descriptions of the required hardware changes are provided in our previous paper [8].

4.5. Advantages and Disadvantages of Wish Branches

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

1. *Wish jumps/joins provide a mechanism to dynamically eliminate the performance and power 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/joins 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 compile-time decision. For this reason, compilers have been conservative in producing predicated code and have avoided large predicated code blocks.
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 [1, 9]. Hence, predicated execution was not applicable for a significant fraction of hard-to-predict branches.
4. *Wish branches will also reduce the need to re-compile the predicated binaries whenever the machine configuration and branch prediction mechanisms change from one processor generation to another (or even during compiler development).* A branch that is hard-to-predict in an older processor may become easy-to-predict in a newer processor with a better branch pre-

dictor. If that branch is conventionally predicated by the old compiler, the performance of the old code will degrade on the new processor because predicated execution would not improve, and in fact degrade, the performance of the now easy-to-predict branch. Hence, to utilize the benefits of the new processor, the old code needs to be recompiled. In contrast, if the branch were converted to a wish branch by the compiler, the performance of the old binary would not degrade on the new processor, since the new processor can dynamically decide not to use predicated execution for the easy-to-predict wish branch. Thus, wish branches reduce the need to frequently re-compile by providing flexibility (dynamic adaptivity) to predication.

The disadvantages of wish branches compared to conventional 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. 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.

5. Performance Evaluation

We have implemented the wish branch code generation algorithm in the state-of-the-art ORC compiler [11]. 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.

The processor we model is eight μops wide and has a 512-entry instruction window, 30-stage pipeline, 64KB two-cycle I-cache; 64KB two-cycle D-cache, 1MB six-cycle unified L2 cache, and 300-cycle minimum main memory latency. We model a very large and accurate hybrid branch predictor (64K-entry gshare/PAs hybrid) and a 1KB JRS confidence estimator [10]. Less aggressive out-of-order processors are also evaluated in our previous paper [8].

We use two predicated code binaries (PRED-SEL, PRED-ALL) as our baselines because neither binary performs the best for all benchmarks. In the PRED-SEL binary, branches are selectively predicated based on cost-benefit analysis. In the PRED-ALL binary, *all* branches suitable for if-conversion are converted to predicated code. Hence, the PRED-ALL binary contains more aggressively predicated code. A wish branch binary contains wish branches, traditional predicated code, and normal branches. Very simple heuristics were used to decide which branches were to be converted to wish branches. Our detailed experimental methodology and heuristics are explained in [8].

5.1. Results

Figure 4 shows the performance of wish branches when wish jumps/joins and loops are utilized. Execution times are normalized to normal branch binaries (i.e., non-predicated binaries). 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 (PRED-SEL). An improved confidence estimator has the potential to increase the performance improvement up to 16.2% over the normal branch binaries. Even if mcf, which skews the average, 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 code binaries (PRED-ALL), with a real confidence estimator.

We also compared 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, PRED-SEL binary, and PRED-ALL 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.

Table 1 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.

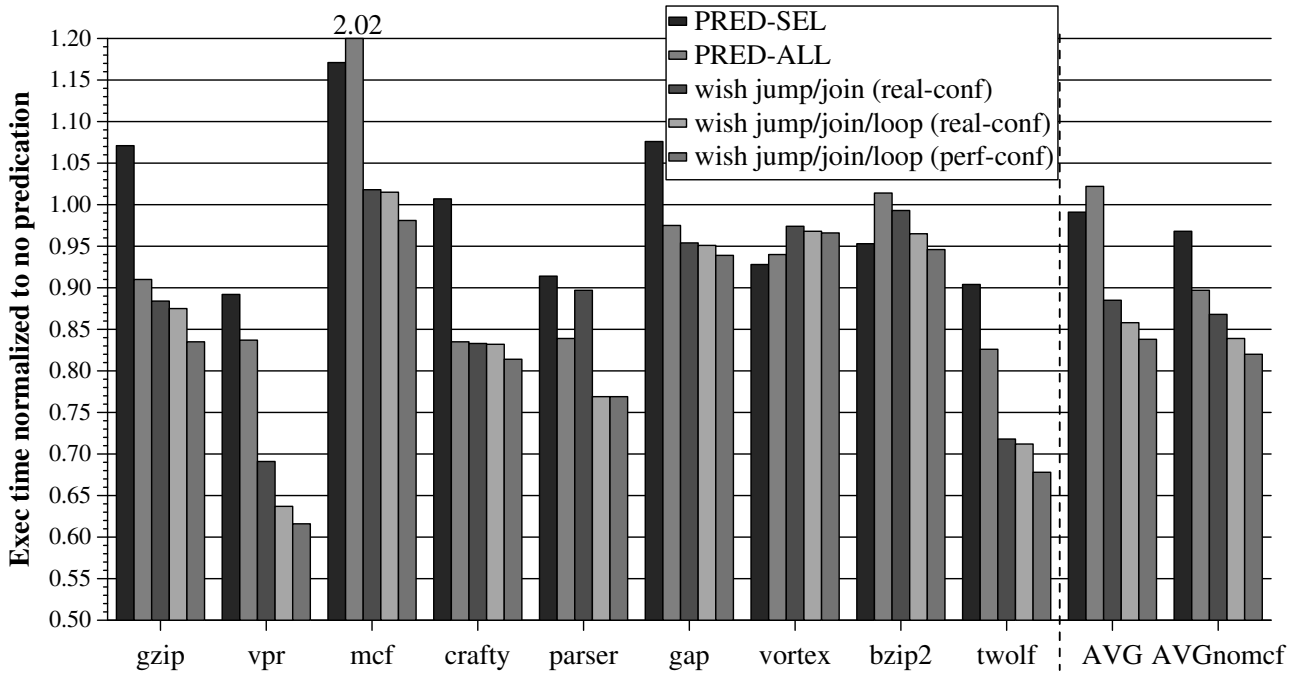


Figure 4. Performance of wish branches.

Table 1. 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). SEL (PRED-SEL), ALL (PRED-ALL), 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% ALL	23.9% ALL	13.3% SEL	0.4% ALL	8.3% ALL	2.5% ALL	-4.3% SEL	-1.2% SEL	13.8% ALL	6.7%
3	% exec time reduction vs. the best non-wish-branch binary for the benchmark	3.8% ALL	23.9% ALL	-1.5% BR	0.4% ALL	8.3% ALL	2.5% ALL	-4.3% SEL	-1.2% SEL	13.8% ALL	5.1%

6. Pending and Future Work

6.1. Compiler Optimizations

The next step in our research is to develop compiler algorithms and heuristics to decide which branches should be converted to wish branches. For example, an input-dependent branch, whose accuracy varies significantly with the input data set of the program, is the perfect candidate to be converted to a wish branch. Since an input-dependent branch is sometimes easy-to-predict and sometimes hard-to-predict depending on the input set, the compiler is more apt to convert such a branch to a wish branch rather than predicating it or leaving it as a normal branch. Similarly, if the compiler can identify branches whose prediction accuracies significantly change depending on the program phase or the control-flow path leading to the branch, it would be more apt to convert them into wish branches.

Our current work, *2D-profiling* [12], can identify input-dependent branches by profiling with only one input set. We call our mechanism 2D-profiling because the profiling compiler collects profile information in two dimensions during the profiling run: *prediction accuracy of a branch* over *time*. If the prediction accuracy of the branch changes significantly during the profiling run with a single input data set, then the compiler predicts that its prediction accuracy will also change significantly across input sets. We have found that 2D-profiling works well because branches that show phased behavior in prediction accuracy tend to be input-dependent.

Other compile-time heuristics or profiling mechanisms that would lead to higher-quality wish branch code are also an area of future work. For example, if the compiler can identify that converting a branch into a wish branch will significantly reduce code optimization opportunities as opposed to predicating it, it could be better off predicating the branch. This optimization would eliminate the cases where wish branch code performs worse than conventionally predicated code due to reduced scope for code optimization, such as for the benchmark vortex as shown in Table 1.

Similarly, if the compiler can take into account the execution delay due to the data dependencies on predicates when estimating the execution time of wish branch code on

an out-of-order processor, it can perform a more accurate cost-benefit analysis to determine what to do with a branch. Such heuristics will also be useful in generating better predicated code for out-of-order processors.

6.2. Hardware Optimizations

On the hardware side, more accurate confidence estimation mechanisms are interesting to investigate since they would increase the performance benefits of wish branches as we have shown in Figure 4. A specialized hardware wish loop predictor could also increase the benefits of wish loops.

7. Conclusion

Wish branches improve performance by dividing the work of predication between the compiler and the microarchitecture. The compiler does what it does best: analyzing the control-flow graphs and producing predicated code, and the microarchitecture does what it does best: making runtime decisions (as to whether or not to use predicated execution or branch prediction for a particular dynamic branch) based on dynamic program information unavailable to the compiler.

This division of work between the compiler and the microarchitecture enables higher performance without a significant increase in hardware complexity. As current processors are already facing power and complexity constraints, wish branches can be an attractive solution to reduce the branch misprediction penalty in a simple and power-efficient way. Hence, wish branches can make predicated execution more viable and effective in future high performance processors.

Acknowledgments

We thank David Armstrong, Robert Cohn, Hsien-Hsin S. Lee, HP TestDrive, Roy Ju, Derek Chiou, and the members of the HPS research group. 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 *10th ACM Symposium on Principles of Programming Languages*, pages 177–189, 1983.

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

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

[4] Eric Sprangle and Yale 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.

[5] Perry H. Wang, Hong Wang, Ralph M. Kling, Kalpana Ramakrishnan, and John 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.

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

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

[8] Hyesoon Kim, Onur Mutlu, Jared Stark, and Yale N. Patt. Wish branches: Combining conditional branching and predication for adaptive predicated execution. In *Proceedings of the 38th ACM/IEEE International Symposium on Microarchitecture*, 2005.

[9] Youngsoo Choi, Allan Knies, Luke Gerke, and Tin-Fook 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.

[10] Erik Jacobsen, Eric 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.

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

[12] Hyesoon Kim, Muhammad Aater Suleman, Onur Mutlu, and Yale N. Patt. 2D-profiling: Detecting input-dependent branches with a single input data set. In *the 4th Annual International Symposium on Code Generation and Optimization*, 2006.