
WISH BRANCHES: ENABLING ADAPTIVE AND AGGRESSIVE PREDICATED EXECUTION

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, USE PREDICATION TO REDUCE THE MISPREDICTION PENALTY FOR HARD-TO-PREDICT BACKWARD (LOOP) BRANCHES.

Hyesoon Kim
Onur Mutlu
University of Texas at
Austin

Jared Stark
Intel Corp.

Yale N. Patt
University of Texas at
Austin

..... Predicated execution has been used to avoid performance loss because of 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 runtime (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 runtime behavior of the branch differs from the compile-time profile behavior, the hardware does not have the ability to override the compiler's choice. A predicated branch remains predicated for all its dynamic instances even if it turns out to be very easy to predict at runtime. Though 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 want to eliminate the performance loss resulting from predicated execution's overhead by letting the hardware choose 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 runtime information. In contrast, the hardware has access to accurate runtime information about each branch.

We propose a mechanism in which the compiler generates code that can be executed either as predicated code or nonpredicated code (that is, code with normal conditional branches). The hardware decides whether to execute the predicated or nonpredicated code based on a runtime confidence estimation of the branch's prediction. The compiler-generated code 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 to avoid 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 branch prediction and predicated execution, depending on accurate runtime information about the branch's behavior.

Predicated execution overhead

Because it converts control dependencies into data dependencies, predicated execution introduces two major sources of overhead in the dynamic execution of a program, which do not occur in conditional branch prediction. First, the processor must fetch additional instructions that are guaranteed to be useless because their predicates will be "false." These instructions waste fetch and possibly execution bandwidth, and occupy processor resources that useful instructions could otherwise use. Second, an instruction that depends on a predicate value cannot execute until the predicate value it depends on is ready. This introduces additional delay into the execution of predicated instructions and their dependents, and thus may increase the program's execution time. In our previous paper,² we analyzed the performance impact of these two overhead sources on an out-of-order processor model that implements predicated execution. We showed that these two overhead sources, especially the execution delay because of predicated instructions, significantly reduce the performance benefits of predicated execution. When we model the overhead of predicated execution faithfully, the predicated binaries do not improve the average execution time of a set of SPEC CPU2000 integer benchmarks. In contrast, if we ideally eliminate all overhead, the predicated binaries would provide 16.4 percent improvement in average execution time.

In addition, one of the limitations of predicated execution is that it cannot eliminate all

Microarchitectural support for predicated execution in out-of-order execution processors

Processor designers implemented predicated execution in in-order processors,^{1,2} but the technique can be used in out-of-order processors as well.^{3,4} 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. Here, 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 might or might not write into its destination register, depending on the value of the predicate.³ Researchers have proposed several solutions to handle this problem: converting predicated instructions into C-style conditional expressions,³ breaking predicated instructions into two micro-ops,⁵ using the select-micro-op mechanism,⁴ and using predicate prediction.⁶ We briefly describe our baseline mechanism, C-style conditional expressions. Our previous paper also evaluates the select-micro-op mechanism.⁷

Our baseline mechanism transforms a predicated instruction into another instruction that is similar to a C-style conditional expression. For example, we convert the $(p1) r1 = r2 + r3$ instruction to the micro-op $r1 = p1 ? (r2 + r3):r1$. 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 no-op operation. Hence, regardless of the predicate value, the instruction always writes into the destination register, allowing the correct renaming of dependent instructions. This mechanism requires four register sources (the old destination register value, the source predicate register, and the two source registers).

References

1. B.R. Rau et al., "The Cydra 5 Departmental Supercomputer," *Computer*, vol. 22, no. 1, Jan. 1989, pp. 12-35.
2. *IA-64 Intel Itanium Architecture Software Developer's Manual Volume 3: Instruction Set Reference*, Intel Corp., 2002; <http://www.intel.com/design/itanium2/documentation.htm>.
3. E. Sprangle and Y. Patt, "Facilitating Superscalar Processing via a Combined Static/Dynamic Register Renaming Scheme," *Proc. 27th ACM/IEEE Int'l Symp. Microarchitecture (Micro-27)*, IEEE CS Press, 1994, pp. 143-147.
4. P.H. Wang et al., "Register Renaming and Scheduling for Dynamic Execution of Predicated Code," *Proc. 7th IEEE Int'l Symp. High Performance Computer Architecture (HPCA 7)*, IEEE CS Press, 2001, pp. 15-26.
5. *Alpha 21264 Microprocessor Hardware Reference Manual*, Compaq Computer Corp., 1999; <http://ftp.digital.com/pub/Digital/info/semiconductor/literature/dsc-library.html>.
6. W. Chuang and B. Calder, "Predicate Prediction for Efficient Out-of-Order Execution," *Proc. 17th Int'l Conf. on Supercomputing (ICS 03)*, ACM Press, 2003, pp. 183-192.
7. H. Kim et al., "Wish Branches: Combining Conditional Branching and Predication for Adaptive Predicated Execution," *Proc. 38th ACM/IEEE Int'l Symp. Microarchitecture (Micro-38)*, IEEE CS Press, 2005, pp. 43-54.

branches. For example, backward (loop) branches, which constitute a significant proportion of all branches, cannot be eliminated using predicated execution.^{1,3}

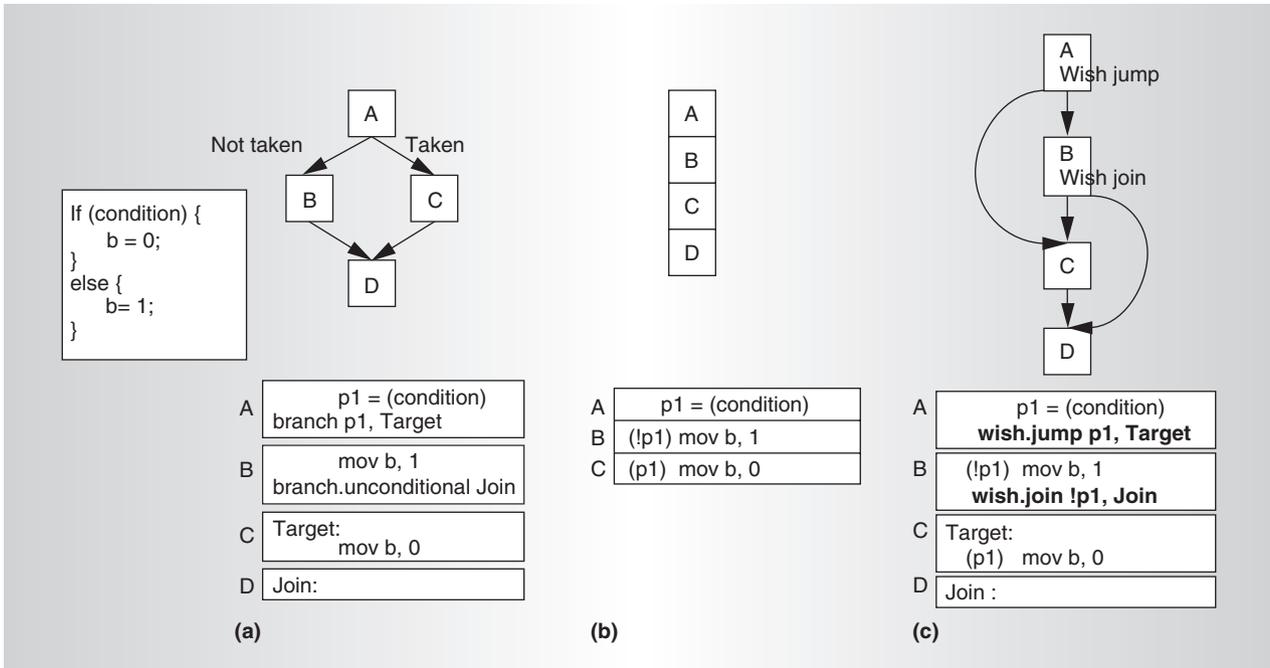


Figure 1. Source code and the corresponding control flow graphs and assembly code for normal branch (a), predicated (b), and wish jump/join (c) codes.

In this article, we propose the use of wish branches to dynamically reduce the overhead sources in predicated execution and to make predicated execution’s benefits applicable to backward branches. These improvements would increase the viability and effectiveness of predicated execution in high-performance, out-of-order execution processors (for background, see sidebar, “Microarchitectural support for predicated execution in out-of-order execution processors”).

Wish jumps and wish joins

Figure 1 shows a simple source code example and the corresponding control flow graphs as well as assembly code for a normal branch, predicated execution, and 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 (as Figure 1c shows), but they are not predicated in the normal branch code (Figure 1a). The first conditional branch in the normal branch code becomes a wish jump instruction, and the following control-dependent unconditional branch becomes 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 (wish jump and wish join), but the predicated code does not.

Wish jump/join code can execute in two different modes (high-confidence mode and low-confidence mode) at runtime. 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 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; in this mode the processor predicts the wish jump instruction 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 processor predicts the wish jump to be taken,

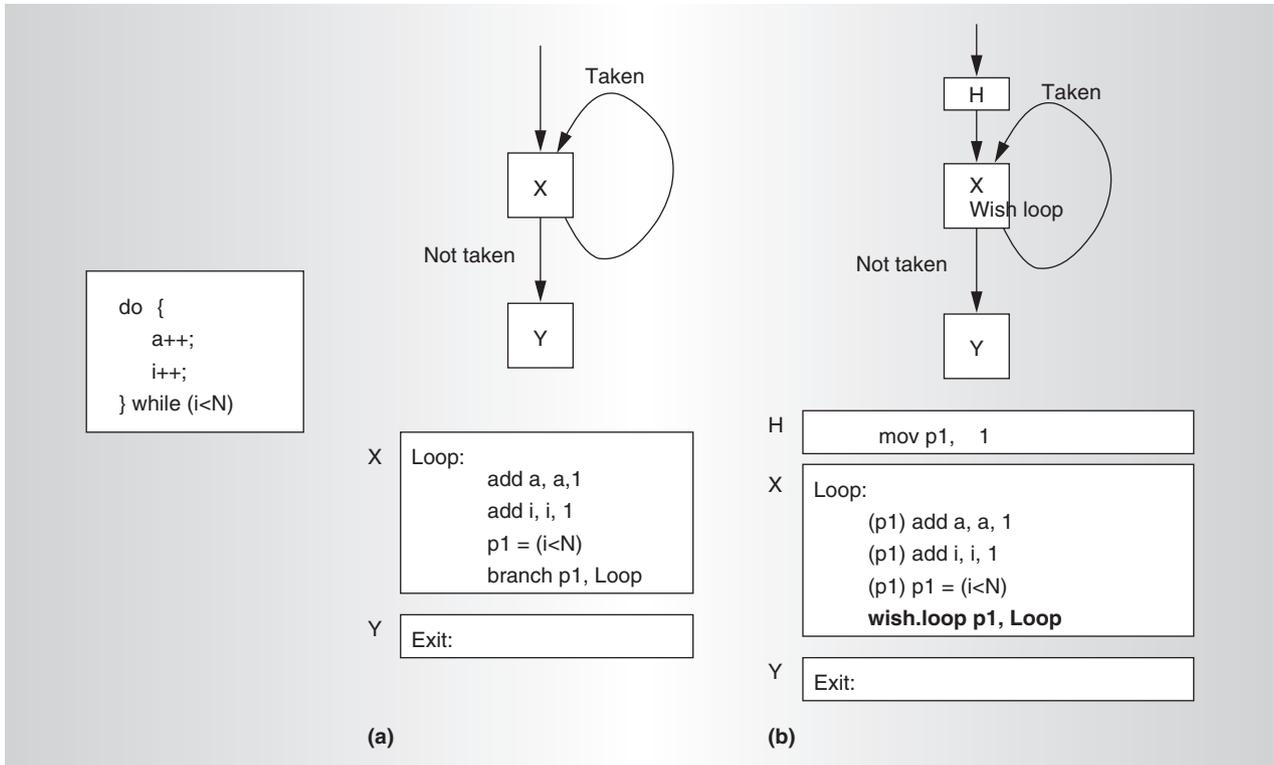


Figure 2. Do-while loop source code and the corresponding control flow graphs and assembly code for normal backward branch (a) and wish loop (b) codes.

it sets the predicate value to true (and does not fetch block B, which contains the wish join). When it predicts the wish jump to be not taken, it sets the predicate value to be false and it predicts the wish join 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 processor does not predict the source predicate value of the wish jump instruction, and the instructions that depend on the predicate only execute when the predicate value is ready.

When the confidence estimate for the wish jump is accurate, the processor either avoids the overhead of predicated execution (high-confidence mode) or eliminates a branch misprediction (low-confidence mode). When the processor mispredicts the wish jump in high-confidence mode, it needs to flush the pipeline just as 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 conventional predicated code, the instructions that are not on the correct control flow path will become no-ops because all instructions that are control-dependent on the branch are predicated by the compiler.

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 control flow graphs and assembly code for a normal backward branch and a wish loop. We compare wish loops only with normal branches since predication cannot directly eliminate backward branches.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 (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 use a wish loop instruction.

When it first encounters the wish loop instruction, the processor enters either high- 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 with the loop/branch predictor. If it predicts the wish loop to be taken, it also predicts the predicate value ($p1$ in Figure 2b) 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 it exits the loop. In low-confidence mode, the processor still predicts the wish loop with the loop/branch predictor. However, it does not predict the predicate value. Hence, in low-confidence mode, the processor executes the loop iterations as predicated code (that is, the processor fetches them but does not execute them until the predicate value is known). There are three misprediction cases in this mode:

- *Early exit.* The loop iterates fewer times than it should.
- *Late exit.* The loop iterates only a few more times than it should, and the front end has already exited the loop when the wish loop misprediction is signaled.
- *No exit.* The loop is still iterating when the wish loop misprediction is signaled (as in the late-exit case, the loop iterates more times than necessary).

For example, consider a loop that iterates three times. The correct loop branch directions are TTN (taken, taken, not taken) for the three iterations, and the front end must fetch blocks $X_1X_2X_3Y$, where X_i is the i th iteration of the loop body. An example for each of the three misprediction cases is as follows.

In the early-exit case, the predictions for the loop branch are TN, so the processor front end fetches blocks X_1X_2Y . One example of the

late-exit case is when the predictions for the loop branch are TTTTN so the front end fetches blocks $X_1X_2X_3X_4X_5Y$. For the no-exit case, the predictions for the loop branch are TTTTTT ... T so the front end fetches blocks $X_1X_2X_3X_4X_5 \dots X_N$.

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

In the late-exit case, 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 no-ops (with predicate value $p1 = \text{false}$) 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 $X_4X_5 \dots X_N$ become no-ops as in the late-exit case, but that would increase energy consumption without improving performance.

Wish branches in complex control flow

Wish branches are not only 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 corresponding normal branch, predicated, and wish branch codes.

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 depends on the confidence estimations made for the previous wish jump, any

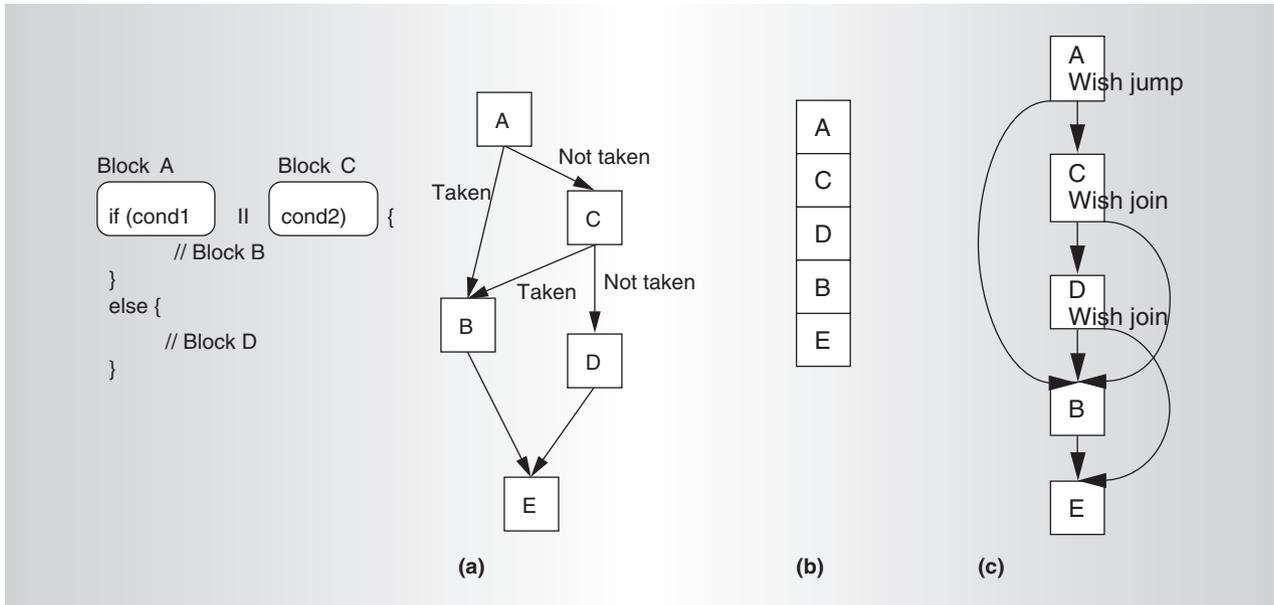


Figure 3. A complex control flow graph example with wish branches: Normal branch (a), predicated (b), and wish branch codes (c).

previous wish joins, and the current wish join itself. If the previous wish jump, any of the previous wish joins, or the current wish join is low-confidence, the current wish join is predicted to be not taken. Otherwise, the current wish join is predicted using the branch predictor.

Support for wish branches

Since wish branches are an instruction set architecture (ISA) construct, they require support from the ISA, the compiler, and the hardware.

ISA support

We assume that the baseline ISA supports predicated execution. Wish branches are implementable 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.

Compiler support

The compiler needs to support the wish branch code generation algorithm. The algorithm decides which branches to predicate, which to convert to wish branches, and which to keep as normal branches based on estimated branch misprediction rates, compile-time

heuristics, and information about branch behavior.²

Hardware support

An accurate confidence estimator⁴ 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. Our previous paper provides detailed descriptions of the required hardware changes.²

Advantages and disadvantages of wish branches

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

- *Wish jumps/joins provide a mechanism to dynamically eliminate the performance 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 runtime confidence estimation of the branch's prediction.
- *Wish jumps/joins allow the compiler to generate predicated code more aggressively and using simpler heuristics, since the processor can correct the poor compile-time decisions*

at runtime. In previous research, a static branch instruction either remained a conditional branch or was predicated for all its dynamic instances, based on less accurate compile-time information; if the compiler made a poor decision to predicate, there was no way to dynamically eliminate the overhead of this poor compile-time decision. For this reason, compilers have been conservative in producing predicated code and have avoided large predicated code blocks.

- *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 solely using predicated execution.^{1,3} Hence, predicated execution was not applicable to a significant fraction of hard-to-predict branches.
- *Wish branches will also reduce the need to recompile 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 might become easy to predict in a newer processor with a better branch predictor. If an old compiler conventionally predicates that branch, the performance of the old code will degrade on the new processor because predicated execution would not improve but in fact would degrade the performance of the now easy-to-predict branch. Hence, to get the benefits of the new processor, the old code would have to be recompiled. In contrast, if the compiler converts such a branch to a wish branch, the old binary's performance 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 recompile by providing flexibility (dynamic adaptivity) to predication.

The disadvantages of wish branches compared to conventional predication are as follows:

- *Wish branches require extra branch instructions.* These instructions would consume machine resources and instruction cache space. However, the larger the predicated code block, the less significant this becomes.
- *The extra wish branch instructions increase the contention for branch predictor table entries.* This might increase negative interference in the pattern history tables. We found that performance loss due to this effect is negligible.
- *Wish branches reduce the size of the basic blocks by adding control dependencies into the code.* Larger basic blocks can provide more opportunities for compiler optimizations. If the compiler that generates wish branch binaries is unable to perform aggressive code optimizations across basic blocks, the presence of wish branches might constrain the compiler's scope for code optimization.

Performance evaluation

We have implemented the wish branch code generation algorithm in the state-of-the-art Open Research Compiler (ORC).⁵ 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-ops to execute on our out-of-order superscalar processor model.

The processor we model is eight micro-ops wide and has a 512-entry instruction window, 30-stage pipeline, 64-Kbyte two-cycle instruction cache, 64-Kbyte two-cycle data cache, 1-Mbyte six-cycle unified L2 cache, and a 300-cycle-minimum main-memory latency. We model a very large and accurate hybrid branch predictor (a 64K entry, gshare/Per Address (PA) hybrid) and a 1 Kbyte confidence estimator.⁴ Our previous paper also evaluates less aggressive out-of-order processors.²

We use two predicated code binaries (PRED-SEL and PRED-ALL) as our baselines because neither binary performs the best for all benchmarks. The compiler selectively predicates branches based on a cost-benefit analysis to produce the PRED-SEL binary. The compiler converts *all* branches suitable for if-conversion to predicated code to produce the PRED-ALL binary. Hence, the PRED-ALL binary contains more aggressively predicated code. A wish

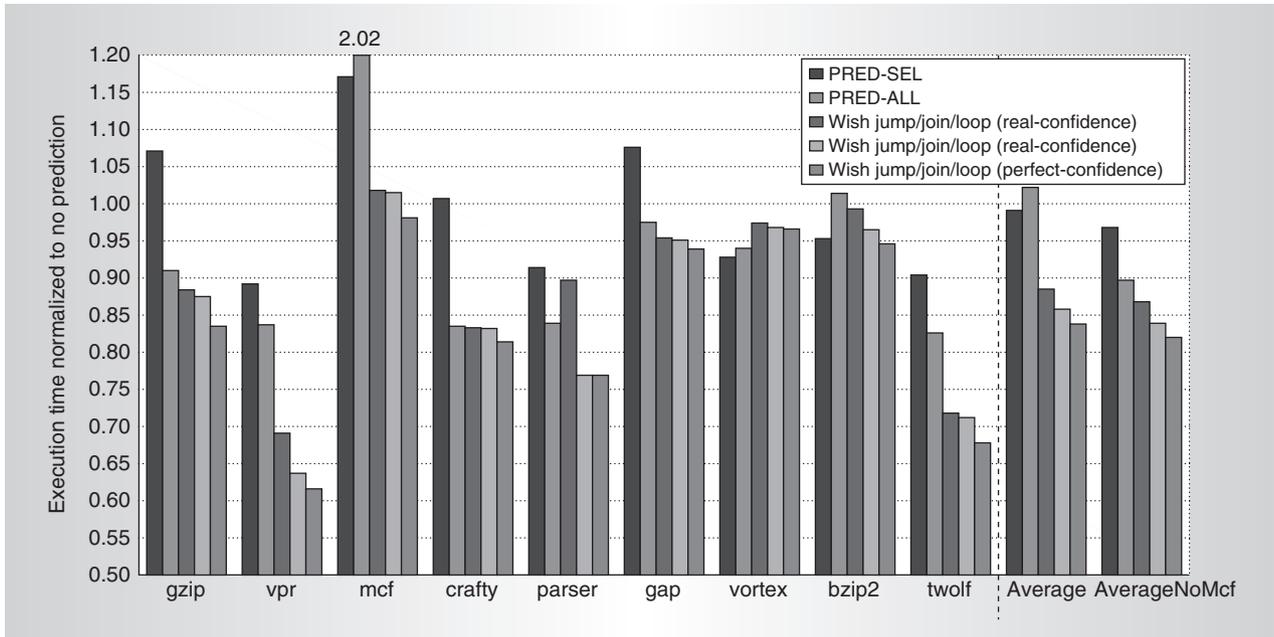


Figure 4. Wish branch performance.

branch binary contains wish branches, traditional predicated code, and normal branches. We used very simple heuristics to decide which branches to convert to wish branches. Our previous paper explains the detailed experimental methodology and heuristics.²

Results

Figure 4 shows the performance of wish branches when the code uses wish jumps, joins, and loops. We normalized execution times with respect to normal branch binaries (that is, non-predicated binaries). With a real confidence estimator, the binaries using wish jumps, joins, and loops (wish branch binaries) improve the average execution time by 14.2 percent compared to the normal branch binaries and by 13.3 percent 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 percent over the performance of the normal branch binaries. Even if we exclude mcf, which skews the average, from the calculation of the average execution time, the wish branch binaries improve the average execution time by 16.1 percent compared to the normal branch binaries and by 6.4 percent 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 from 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. 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 runtime. This assumption is not correct because the compiler does not “know” the runtime behavior of the branches in the program. Even worse, the runtime 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 branch binary compared to the normal branch binary (column 2), the best-performing predicated code binary for the benchmark (column 3), and the best-performing binary (that does not contain wish branches) for the benchmark (column 5). Even if the compiler were able to choose and generate the best-performing binary for each benchmark, the wish branch binary outperforms the best-

Table 1. Execution time reduction percentage of the wish branch binaries over the best-performing binaries on a per-benchmark basis (using the real confidence mechanism).*

Execution time reduction of the wish jump/join/loop binaries versus other binaries (percentage)

| Benchmark | Normal branch | Best predicated code | Best predicated code Binary* | Best non-wish-branch | Best non-wish-branch Binary* |
|------------------|----------------------|-----------------------------|-------------------------------------|-----------------------------|-------------------------------------|
| gzip | 12.5 | 3.8 | PRED-ALL | 3.8 | PRED-ALL |
| vpr | 36.3 | 23.9 | PRED-ALL | 23.9 | PRED-ALL |
| mcf | -1.5 | 13.3 | PRED-SEL | -1.5 | Branch |
| crafty | 16.8 | 0.4 | PRED-ALL | 0.4 | PRED-ALL |
| parser | 23.1 | 8.3 | PRED-ALL | 8.3 | PRED-ALL |
| gap | 4.9 | 2.5 | PRED-ALL | 2.5 | PRED-ALL |
| vortex | 3.2 | -4.3 | PRED-SEL | -4.3 | PRED-SEL |
| bzip2 | 3.5 | -1.2 | PRED-SEL | -1.2 | PRED-SEL |
| twolf | 29.8 | 13.8 | PRED-ALL | 13.8 | PRED-ALL |
| average | 14.2 | 6.7 | NA | 5.1 | NA |

* PRED-SEL, PRED-ALL, and Branch (normal branch) indicate which binary is the best performing binary for a given benchmark.

performing binary for each benchmark by 5.1 percent on average, as column 5 shows.

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.

The next step in our research is to develop compiler algorithms and heuristics to decide which branches to convert to wish branches. For example, an input-dependent branch, whose accuracy varies significantly with the program's input data set, is the per-

fect candidate for conversion 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 change significantly, 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.

We have devised a mechanism for identifying input-dependent branches by profiling with only one input set. We call our mechanism 2D-profiling⁶ 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 iden-

tify that converting a branch into a wish branch (as opposed to predicating it) will significantly reduce code optimization opportunities, it could be better off predicating the branch. This optimization would eliminate the cases where wish branch code performs worse than conventionally predicated code because of the reduced scope for code optimization, such as for the vortex benchmark in Table 1.

Similarly, if the compiler can account for the execution delay from 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.

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

MICRO

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 Corp., 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 et al., "Conversion of Control Dependence to Data Dependence," *Proc. 10th ACM SIGACT-SIGPLAN Symp. Principles of Programming Languages (POPL 83)*, ACM Press, 1983, pp. 177-189.
2. H. Kim et al., "Wish Branches: Combining Conditional Branching and Predication for Adaptive Predicated Execution," *Proc. 38th ACM/IEEE Int'l Symp. Microarchitecture (Micro-38)*, IEEE CS Press, 2005, pp. 43-54.
3. Y. Choi et al., "The Impact of If-Conversion and Branch Prediction on Program Execution on the Intel Itanium Processor," *Proc. 34th ACM/IEEE Int'l Symp. Microarchitecture (Micro-34)*, IEEE CS Press, 2001, pp. 182-191.
4. E. Jacobsen, E. Rotenberg, and J.E. Smith, "Assigning Confidence to Conditional Branch Predictions," *Proc. 29th ACM/IEEE Int'l Symp. Microarchitecture (Micro-29)*, IEEE CS Press, 1996, pp. 142-152.
5. ORC, "Open Research Compiler for Itanium Processor Family," <http://ipf-orc.sourceforge.net/>.
6. H. Kim et al., "2D-Profiling: Detecting Input-Dependent Branches with a Single Input Data Set," to appear in *Proc. 4th Annual International Symposium on Code Generation and Optimization (CGO 4)*, 2006.

Hyesoon Kim is a PhD candidate in electrical and computer engineering at the University of Texas at Austin. Her research interests include high-performance energy-efficient microarchitectures and compiler-microarchitecture interaction. Kim has master's degrees in mechanical engineering from Seoul National University, and in computer engineering from UT Austin. She is a student member of the IEEE and the ACM.

Onur Mutlu is a PhD candidate in computer engineering at the University of Texas at Austin. His research interests include computer architectures, with a focus on high-performance energy-efficient microarchitectures, data prefetching, runahead execution, and novel latency-tolerance techniques. Mutlu has an MS in computer engineering from UT Austin and BS degrees in psychology and computer engineering from the University of Michigan. He is a student member of the IEEE and the ACM.

Yale N. Patt is the Ernest Cockrell, Jr. Centennial Chair in Engineering at the University of Texas at Austin. His research interests include harnessing the expected fruits of future process technology into more effective microarchitectures for future microprocessors. He is co-author of *Introduction to Computing Systems: From Bits and Gates to C and Beyond* (McGraw-Hill, 2nd edition, 2004). His honors include the 1996 IEEE/ACM Eckert-Mauchly Award and the 2000 ACM Karl V. Karlstrom Award. He is a Fellow of both the IEEE and the ACM.

Jared Stark is a computer architect at Intel's Israel Development Center in Haifa, Israel.

His research interests include front-end microarchitecture, in particular, branch prediction; dynamic instruction scheduling; and techniques for tolerating cache and memory latencies. Stark has a BSE in electrical engineering and MSE and PhD in computer science and engineering from the University of Michigan, Ann Arbor. He is a member of the IEEE.

Direct questions and comments about this article to Hyesoon Kim, Department of Electrical and Computer Engineering, University of Texas at Austin, 1 University Station C0803, Austin, TX 78712; hyesoon@ece.utexas.edu.

For further information on this or any other computing topic, visit our Digital Library at <http://www.computer.org/publications/dlib>.

ADVERTISER / PRODUCT INDEX JANUARY / FEBRUARY 2006

FUTURE ISSUES

March/April 2006:

Hot Chips 17

May/June 2006:

High-Performance On-Chip Interconnects

July/August 2006:

Computer Architecture Simulation and Modeling

Advertising Personnel

Marian Anderson

Advertising Coordinator

Phone: +1 714 821 8380

Fax: +1 714 821 4010

Email: manderson@computer.org

Sandy Brown

IEEE Computer Society,

Business Development Manager

Phone: +1 714 821 8380

Fax: +1 714 821 4010

Email: sb.ieeemedia@ieee.org

For production information, conference and classified advertising, contact Marian Anderson, *IEEE Micro*, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-1314; phone +1 714 821 8380; fax +1 714 821 4010; manderson@computer.org.

<http://www.computer.org>