Diverge-Merge Processor: Generalized and Energy-Efficient Dynamic Predication

THE BRANCH MISPREDICTION PENALTY IS A MAJOR PERFORMANCE LIMITER AND A MAJOR CAUSE OF WASTED ENERGY IN HIGH-PERFORMANCE PROCESSORS. THE DIVERGE-MERGE PROCESSOR REDUCES THIS PENALTY BY DYNAMICALLY PREDICATING A WIDE RANGE OF HARD-TO-PREDICT BRANCHES AT RUNTIME IN AN ENERGY-EFFICIENT WAY THAT DOESN'T SIGNIFICANTLY INCREASE HARDWARE COMPLEXITY OR REQUIRE MAJOR ISA CHANGES.

Hyesoon Kim
José A. Joao
University of Texas at Austin

Onur Mutlu
Microsoft Research

Yale N. Patt
University of Texas at Austin

Today's high-performance processors use deep pipelines to support high clock frequencies. Some processing cores in near-future chip multiprocessors will likely support a large number of in-flight instructions[1] to extract both memory- and instruction-level parallelism for high performance and energy efficiency in the serial (nonparallelizable) portions of applications.[2] The performance benefit and energy efficiency of using pipelining and supporting a large number of in-flight instructions critically depend on the accuracy of the processor's branch predictor. Even after decades of research in branch prediction, branch predictors remain imperfect. They frequently mispredict hard-to-predict branches, not only limiting performance but also wasting energy.

To avoid pipeline flushes caused by branch mispredictions, researchers have proposed *predication*, a mechanism that converts a control dependency to a set of data dependencies.[3] This is done by removing the branch instruction from the program and adding the branch condition (that is, predicate) to each of the instructions on both true and false paths following the original branch. Thus, the processor fetches instructions from both sides of the original branch, but commits results from only the correct side, as determined by the branch condition.

However, three major problems have prevented the wide use of predication in high-performance processors:

- *ISA changes*. Predication requires major, widespread changes to the instruction set architecture—in particular, the addition of predicate registers and predicated instructions.
- *Lack of adaptivity*. Predication is not adaptive to runtime branch behavior because a statically if-converted branch instruction remains if-converted regardless of whether its instances are hard to predict at runtime. Previous

Published by the IEEE Computer Society

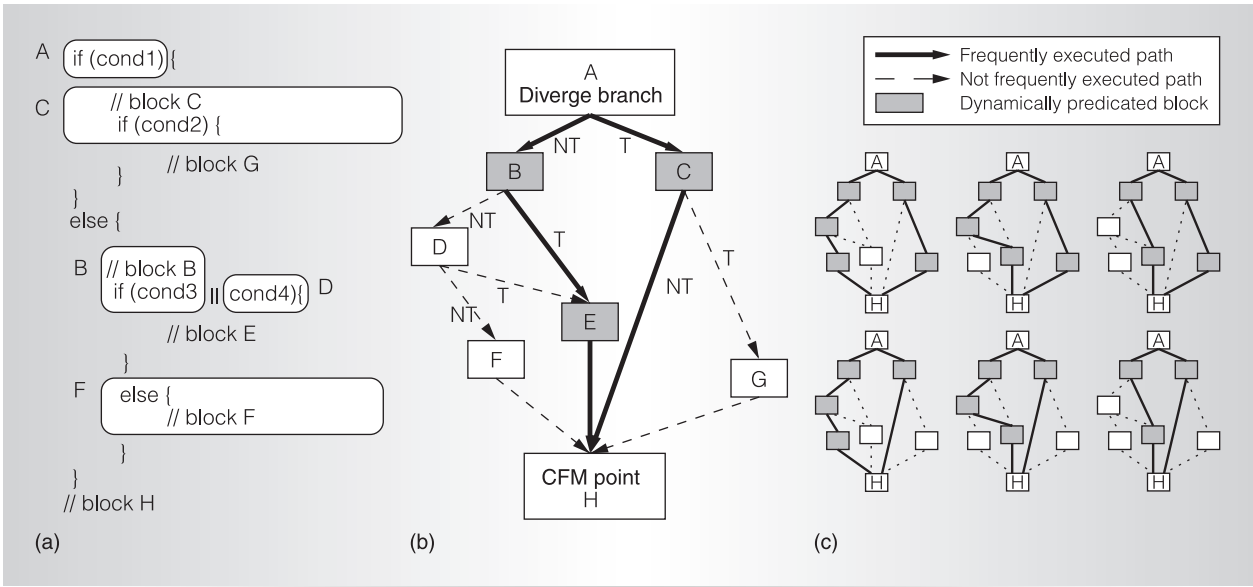

Figure 1. Control-flow graph (CFG) example: source code (a), CFG (b), and possible paths (hammocks) that DMP can predicate (c). CFM: control-flow merge; T: taken; NT: not taken.

research has shown that this lack of adaptivity can significantly reduce predicated code's performance.[4,5]

- *Complex control-flow graphs.* Predication's performance potential is limited because, in many cases, compilers either cannot or usually don't convert control-flow graphs to predicated code. Either these CFGs are too complex or they contain loops, function calls, indirect branches, or too many instructions.[3,4] Current compilers usually don't predicate large and complex CFGs because doing so would cause a large performance overhead.

Researchers have proposed several solutions to these problems. Dynamic hammock predication[6] solves the ISA and adaptivity problems, but it is applicable to only a small set of CFGs.[6,7] Wish branches solve the adaptivity problem and partially solve the complex-CFG problem by enabling the predication of loops, but they still require significant ISA support for predication.[5] Hyperblock formation partially solves the complex-CFG problem.[8] However, no previous approach concurrently solves all three problems. The diverge-merge processor (DMP) we describe in this article provides a comprehensive, energy-efficient solution.

The diverge-merge concept

We propose a new processor architecture that performs predication dynamically on complex CFGs. The key insight that enables DMP to predicate complex CFGs is that most complex control-flow graphs look like simple hammock (if-else) structures if we consider only the frequently executed paths in the graphs. Figure 1 shows a complex-CFG example that illustrates this. In conventional software predication, if the compiler estimates that the branch at block A is hard to predict, it converts blocks B, C, D, E, F, and G to predicated code, and all these blocks will execute together even though blocks D, F, and G are not frequently executed at runtime. In contrast, DMP considers frequently executed paths at runtime, so it can dynamically predicate only blocks B, C, and E, thereby incurring less performance overhead. Moreover, if program phase changes or input set effects cause the frequently executed paths in the complex hammock to change at runtime, DMP can adapt dynamic predication to the newly formed frequently executed paths. In the Figure 1 example, DMP can predicate

many different dynamic hammocks, depending on which paths are frequently executed at runtime (as Figure 1c shows).

Overview of DMP operation

The compiler identifies conditional branches with control flow suitable for dynamic predication as *diverge branches*. A diverge branch is a branch instruction after which program execution usually reconverges at a control-independent point in the CFG called the *control-flow merge (CFM) point*. In other words, diverge branches result in hammock-shaped control flow based on frequently executed paths in the program's CFG, but they are not necessarily simple hammock branches that require the CFG to be hammock shaped (that is, they can form what we call *frequently-hammocks*). The compiler also identifies at least one CFM point associated with the diverge branch. Diverge branches and CFM points can be conveyed to the microarchitecture through relatively simple modifications in the ISA.[7]

When the processor fetches a diverge branch, it estimates whether or not the branch is hard to predict using a branch confidence estimator.[9] If the diverge branch has low confidence, the processor enters dynamic predication (dpred) mode. In this mode, the processor fetches both paths after the diverge branch and dynamically predicates instructions between the diverge branch and the CFM point. On each path, the processor follows the branch predictor outcomes until it reaches the CFM point. After the processor reaches the CFM point on both paths, it exits dpred mode and starts to fetch from only one path. The processor reconciles the register values produced on the dynamically predicated paths by inserting select-micro-operations (select-μops) to merge the register values produced on both sides of the hammock.[10] If the diverge branch is actually mispredicted, the processor doesn't need to flush its pipeline because instructions on both paths of the branch are already fetched and instructions on the wrong path will become NOPs (no operations) through dynamic predication.

DMP support

Now let's turn to the hardware changes required to support DMP. Our previous publication provides a detailed implementation and analyzes its complexity.[7]

Instruction fetch

In dpred mode, the processor fetches instructions from both directions (taken and not-taken paths) of a diverge branch, using two program counter registers and a round-robin scheme to fetch from the two paths in alternate cycles. On each path, the processor follows the branch predictor's outcomes. These outcomes favor the frequently executed basic blocks in the CFG.

The processor uses a separate global branch history register to predict the next fetch address on each path and checks whether the predicted next fetch address is the diverge branch's CFM point. If the processor reaches the CFM point on one path, it stops fetching from that path and fetches only from the other path. When the processor reaches the CFM point on both paths, it exits dpred mode.

Select-μops

Instructions after the CFM point should have data dependencies on instructions only from the diverge branch's correct path. Before executing the diverge branch, the processor doesn't know which path is correct. Instead of waiting for the diverge branch's resolution, the processor inserts select-μops to continue renaming and execution after exiting dpred mode. Select-μops are similar to ϕ-functions in the static single-assignment form in that they merge the register values produced on both sides of the hammock. Select-μops ensure that instructions dependent on the register values produced on either side of the hammock are supplied with the correct data values, which depend on the diverge branch's correct direction.

After inserting select-μops, the processor can continue fetching and renaming instructions. If an instruction fetched after the CFM point is dependent on a register produced on either side of the hammock, it sources a select-μop's output. The processor will execute such an instruction after the

diverge branch resolves. However, the processor executes instructions that are not dependent on select-µops as soon as their sources are ready, without waiting for resolution of the diverge branch. Figure 2 shows an example of the dynamic predication process. Note that the processor executes instructions in blocks C, B, and E, fetched during dpred mode, without waiting for the diverge branch's resolution.

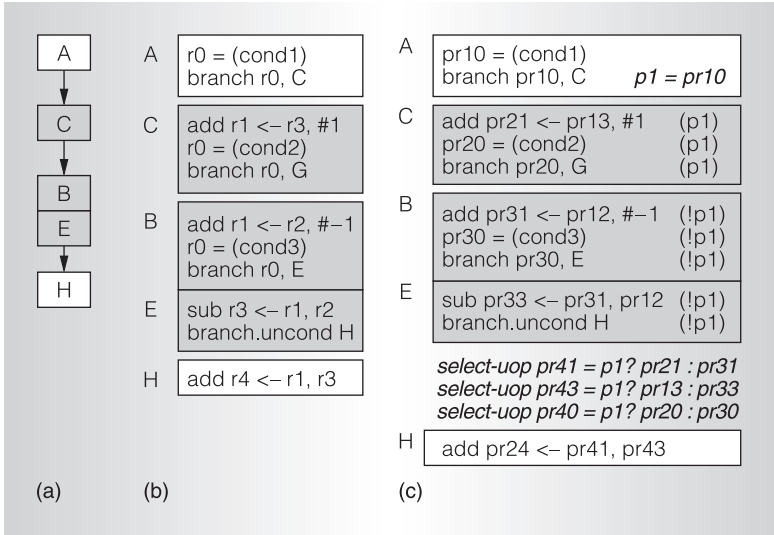

Figure 2. Dynamic predication of the instruction stream in Figure 1b: fetched blocks (a), fetched assembly instructions (b), and instructions after register renaming (c).

Loop branches

DMP can dynamically predicate loop branches. The benefit of this is very similar to the benefit of wish loops, another type of wish branch which is used for handling loops.[5] The key to predicating a loop-type diverge branch is that the processor must predicate each loop iteration separately. It does this by using a different predicate register for each iteration and inserting select-µops after each iteration. Select-µops choose between live-out register values before and after execution of a loop iteration; the choice is based on the outcome of each dynamic instance of the loop branch. Instructions executed in later iterations and dependent on live-outs of previous predicated iterations source the outputs of select-µops. Similarly, instructions fetched after the processor exits the loop and dependent on registers produced within the loop source the select-µop outputs so that they receive the correct source values even though the loop branch might be mispredicted. The pipeline need not be flushed if a predicated loop iterates more times than it should, because the predicated instructions in the extra loop iterations will become NOPs, and the live-out values from the correct last iteration will propagate to dependent instructions via select-µops.

Instruction execution and retirement

DMP executes dynamically predicated instructions just as it does other instructions (except store-load forwarding). Because these instructions depend on the predicate value only for retirement purposes, the processor can execute them before the predicate value (the diverge branch) is resolved. If the predicate value is known to be *false*, the processor need not execute the instructions or allocate resources for them. Nonetheless, all predicated instructions consume retirement bandwidth. When a predicated-false instruction is ready to retire, the processor simply frees the physical register (along with other resources) allocated for that instruction and doesn't update the architectural state with its results. The predicate register associated with dpred mode is released when the last predicated instruction retires.

Load and store instructions

DMP executes dynamically predicated load instructions in the same manner as normal load instructions. It sends dynamically predicated store instructions to the store buffer with their predicate register IDs. However, a predicated store instruction is not written into the memory system (into the caches) until it is known to be predicated-true. The processor drops all predicated-false store requests. Thus, it requires the store buffer logic to check the predicate register value before sending a store request to the memory system.

DMP requires support in the store-load forwarding logic, which should check not only addresses but also predicate register IDs. The logic can forward from three kinds of stores:

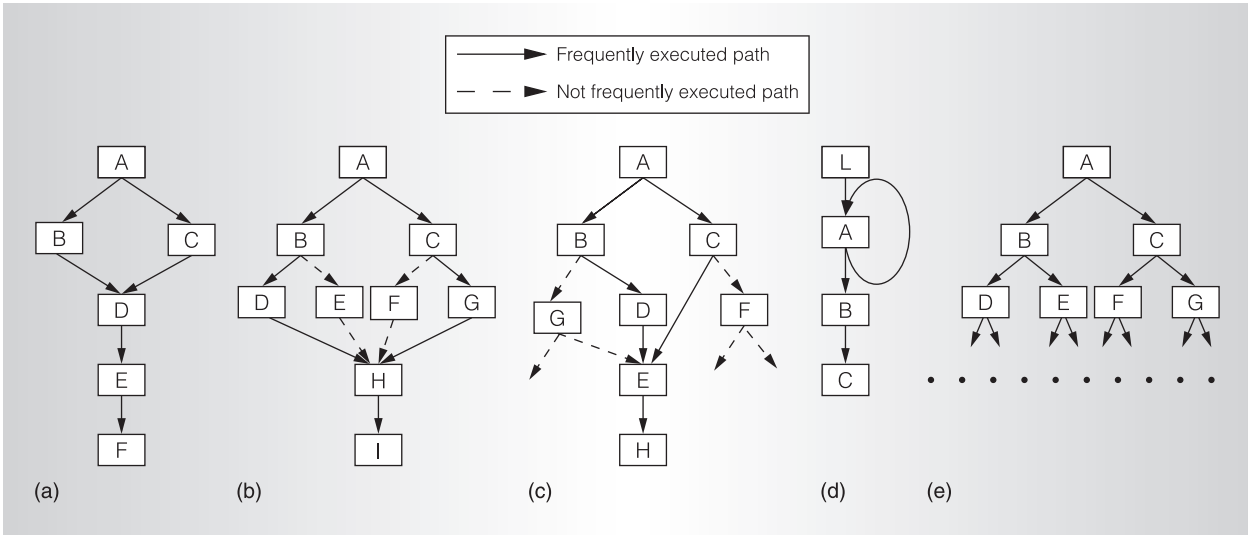

Figure 3. Control-flow graphs: simple hammock (a), nested hammock (b), frequently-hammock (c), loop (d), and nonmerging (e).

- a nonpredicated store to any later load,
- a predicated store whose predicate register value is known to be true to any later load, or
- a predicated store whose predicate register is not ready to a later load with the same predicate register ID (that is, on the same dynamically predicated path).

If forwarding is not possible, the load waits.

This mechanism and its supporting structures are also required in an out-of-order processor that implements any variant of predication (software predication, dynamic hammock predication, or wish branches).

ISA support

Two bits in the ISA's branch instruction format distinguish diverge branches. The first bit indicates whether or not the branch is a diverge branch. The second bit indicates whether or not the branch is a loop branch. The ISA also provides support for encoding the CFM points in the program binary. However, there is no need for ISA support for predicate registers or predicated instructions.

Compiler support

DMP uses a combination of compile-time CFG analysis and profiling to determine diverge branches and CFM points. We describe the compiler heuristics and profiling support for generating DMP executables in another publication.[11]

DMP versus other branch-processing paradigms

Five previously proposed paradigms for handling hard-to-predict branch instructions are *dynamic hammock predication*,[6] *software predication*,[3] *wish branches*,[5] selective *dual-path execution*,[12] and selective *multipath execution*.[13] To illustrate the differences between these mechanisms and compare them with DMP, we classify control-flow graphs into the following categories:

- *simple hammock* (Figure 3a)—an if or if-else structure that has no nested branches inside the hammock;
- *nested hammock* (Figure 3b)—an if-else structure that has multiple levels of nested branches;
- *frequently-hammock* (Figure 3c)—a CFG that becomes a simple hammock if we consider only frequently executed paths;

Table 1. Fetched instructions in various processing models for each CFG type (after the branch at A is estimated to be low-confidence). We assume that the loop branch in block A in Figure 3d is predicted as taken twice after it is estimated to be low-confidence.

Processing model	Simple hammock	Nested hammock	Frequently-hammock	Loop	Nonmerging
DMP	B, C, D, E, F	B, C, D, G, H, I	B, C, D, E, H	A, A, B, C	Can't predicate
Dynamic hammock predication	B, C, D, E, F	Can't predicate	Can't predicate	Can't predicate	Can't predicate
Software predication	B, C, D, E, F	B, C, D, E, F, G, H, I	Usually doesn't or can't predicate	Can't predicate	Can't predicate
Wish branches	B, C, D, E, F	B, C, D, E, F, G, H, I	Usually doesn't or can't predicate	A, A, B, C	Can't predicate
Dual-path	Path 1: B, D, E, F Path 2: C, D, E, F	Path 1: B, D, H, I Path 2: C, G, H, I	Path 1: B, D, E, H Path 2: C, E, H	Path 1: A, A, B, C Path 2: B, C	Path 1: B … Path 2: C …

- *loop* (Figure 3d)—a cyclic CFG (for, do-while, or while structure); and
- *nonmerging* (Figure 3e)—a CFG that has no CFM point even if we consider only frequently executed paths.

Figure 3 depicts these five types of CFGs. Table 1 summarizes the blocks fetched and predicated in five different processing models for each CFG type, assuming that the branch in block A is hard to predict. Figure 4 shows the frequency of branch mispredictions for each CFG type in executing SPEC integer benchmark applications.

Dynamic hammock predication can predicate only simple hammocks, which account for 12 percent of all mispredicted branches. Simple hammocks alone account for a significant percentage of mispredictions in only two benchmarks: vpr (40 percent) and twolf (36 percent). Dynamic hammock predication significantly improves the performance of these two benchmarks.

Software predication can predicate both simple and nested hammocks, which in total account for 16 percent of all mispredicted branches. Software predication

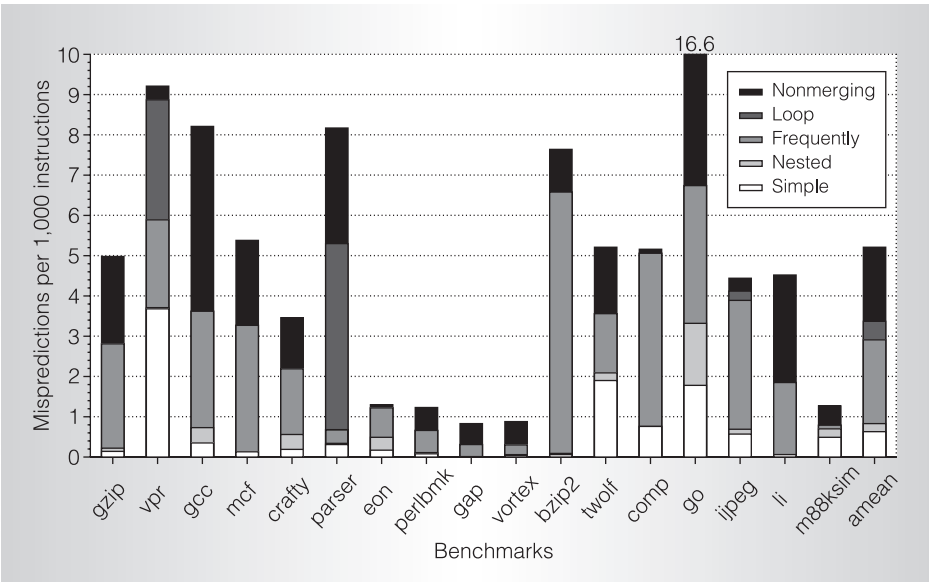

Figure 4. Distribution of mispredicted branches for each CFG type.

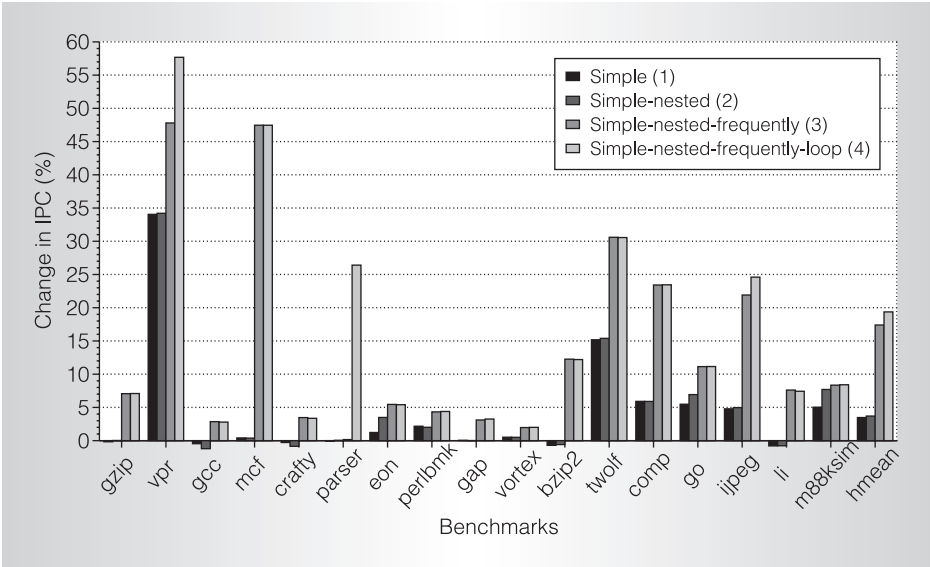

Figure 5. DMP performance when different CFG types are dynamically predicated. IPC: instructions per cycle.

fetches all basic blocks between an if-converted branch and the corresponding CFM point. For example, in the nested-hammock case (Figure 3b), software predication fetches blocks B, C, D, E, F, G, H, and I, whereas DMP fetches blocks B, C, D, G, H, and I. Current compilers usually don't predicate frequently-hammocks because the predicated code's overhead would be too high if these CFGs include function calls, cyclic control-flow, too many exit points, or too many instructions.[3,4,14] Hyperblock formation can predicate frequently-hammocks at the cost of increased code size, but it is not an adaptive technique because frequently executed basic blocks change at runtime.[8] Even if we unrealistically assume that software predication *can* predicate all frequently-hammocks, it would predicate up to 56 percent of all mispredicted branches.

In addition to what software predication can do, wish branches can also predicate loops, which account for another 10 percent of all mispredicted branches. The main difference between wish branches and software predication is that the wish branch mechanism selectively predicates at runtime each dynamic instance of a branch. The wish branch mechanism predicates a branch only if it is hard to predict at runtime, whereas software predication predicates a branch for all its dynamic instances. Thus, wish branches reduce the overhead of software predication. However, even with wish branches, all basic blocks between an if-converted branch and the corresponding CFM point are fetched. Therefore, wish branches also have higher performance overhead for nested hammocks than DMP has.

Software predication (and wish branches) can eliminate a branch misprediction caused by a branch that is control-dependent on another hard-to-predict branch (for example, the branch at B is control-dependent on the branch at A in Figure 3b), because it predicates all the basic blocks within a nested hammock. This benefit is not possible with any of the other paradigms except multi-path, but we found that it provides a significant performance benefit only in two benchmarks (3 percent in twolf and 2 percent in go).

Selective dual-path execution fetches from two paths after a hard-to-predict branch. Instructions on the wrong path are selectively flushed when the branch is resolved. Dual-path execution is applicable to any kind of CFG because the control-flow need not reconverge. Hence, dual-path can potentially eliminate the branch mis-

prediction penalty for all five CFG types. However, the dual-path mechanism requires fetching more instructions than any of the other mechanisms (except multipath) because the processor continues fetching from two paths until the hard-to-predict branch is resolved, even though the processor might already have reached a control-independent point in the CFG. For example, in the simple-hammock case (Figure 3a), DMP fetches blocks D, E, and F only once, but dual-path fetches D, E, and F twice (once for each path). Therefore, dual-path's overhead is much higher than DMP's.

Multipath execution is a generalized form of dual-path execution in that the processor fetches both paths after *every* low-confidence branch and therefore can execute along more than two different paths at the same time. This increases the probability that the correct path is in the processor's instruction window. However, only one of the outstanding paths is the correct path, and instructions on every other path must be flushed. Furthermore, instructions after a control-flow-independent point must be fetched (and perhaps executed) separately for each path (like dual-path, but unlike DMP), wasting processing resources for instructions on all paths but one. For example, if the number of outstanding paths is eight, a multipath processor wastes 87.5 percent of its fetch and execution resources for wrong-path or useless instructions even after a control-independent point. Hence, multipath's overhead is far greater than that of DMP. For the example in Figure 3, multipath's behavior is the same as that of dual-path because, for simplicity, the example assumes only one hard-to-predict branch.

DMP can predicate simple hammocks, nested hammocks, frequently-hammocks, and loops. On average, these four CFG types account for 66 percent of all branch mispredictions. As Table 1 shows, the number of fetched instructions for DMP is less than or equal to the number for the other mechanisms for all CFG types. Hence, DMP eliminates branch mispredictions more efficiently (with less overhead) than the other processing paradigms. Our earlier article on DMP provides detailed comparisons of overhead, performance, hardware complexity, and energy consumption of the different branch-processing paradigms.[7]

Performance evaluation and results

To analyze and evaluate DMP's performance and energy consumption, we used an execution-driven simulator of a processor that implements the Alpha ISA. We modeled an aggressive high-performance eight-wide processor (with a 30-stage pipeline and a 512-entry instruction window) and a less aggressive four-wide processor (with a 20-stage pipeline and a 128-entry instruction window). Our baseline branch predictor is a very aggressive 64-Kbyte perceptron predictor. DMP uses a 2-Kbyte enhanced JRS confidence estimator to detect hard-to-predict branches.[9] We used different input sets for profiling to generate DMP binaries and for performance evaluation. Our earlier article on DMP describes our performance and power simulation and compilation methodology in detail.[7] All of our evaluations quantitatively compare DMP to the five other branch-processing paradigms on all the SPEC CPU2000 and CPU95 integer benchmarks.

Figure 5 shows DMP's performance improvement over the baseline processor when we allowed DMP to dynamically predicate

- simple hammocks only;
- simple and nested hammocks;
- simple, nested, and frequently-hammocks; and
- simple, nested, and frequently-hammocks, plus loops.

Predication of frequently-hammocks provides the largest performance improvement because frequently-hammocks are the single largest cause of branch mispredictions. Hence, DMP provides large performance improvements by enabling the predication of a wide range of complex CFGs.

Figure 6 summarizes the performance, power, and energy comparison of DMP with the other branch-handling paradigms. For all the benchmarks, the average retired

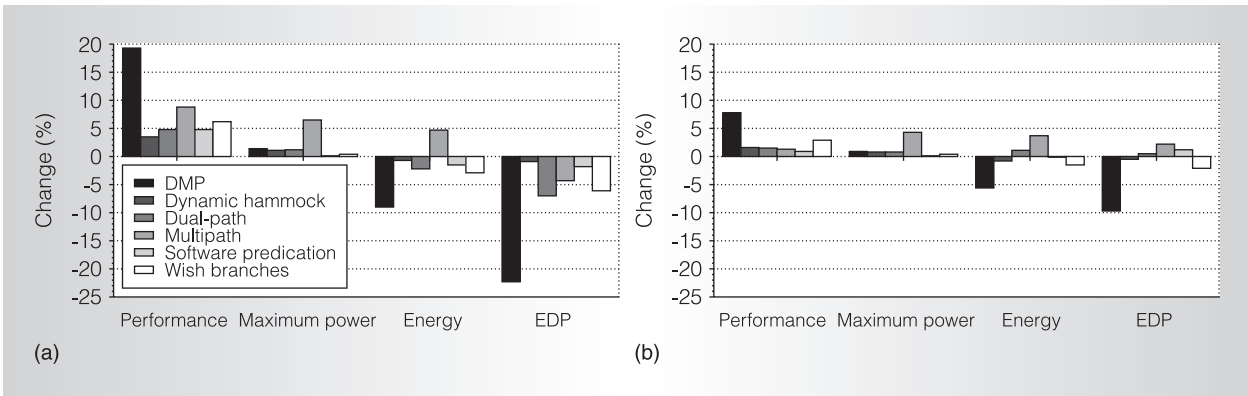

Figure 6. Performance, power, energy, and energy-delay product (EDP) comparison of DMP with other branch-processing paradigms running on aggressive (30-stage, 512-window, eight-wide) (a) and less aggressive (20-stage, 128-window, four-wide) (b) baseline processors. Performance improvements are in terms of instructions per cycle for DMP, dynamic hammock predication, dual-path, and multipath, and in terms of execution time for software predication and wish branches.

instructions per cycle (IPC) performance improvement over the baseline processor is 3.5 percent for dynamic hammock predication,[6] 4.8 percent for dual-path,[12] 8.8 percent for multipath,[13] and 19.3 percent for DMP. Conventional software predication reduces execution time by 3.8 percent, wish branches by 6.4 percent, and DMP by 13.0 percent. DMP provides the best energy efficiency and energy-delay product (EDP) among all paradigms, reducing energy consumption by 9 percent and improving EDP by 22.3 percent. It reduces energy consumption because it reduces the number of pipeline flushes by 38 percent, which results in the fetch of 23 percent fewer instructions (not shown). Even on the less aggressive baseline processor, with a short pipeline and a small instruction window, DMP improves performance by 7.8 percent, reduces energy consumption by 5.6 percent, and improves EDP by 9.7 percent (Figure 6b).

We also evaluated DMP on a baseline with the recently proposed O-Geometric History Length (O-GEHL) branch predictor.[15] Our results show that even on a processor that employs a complex yet very accurate 64-Kbyte O-GEHL predictor, diverge-merge processing improves performance by 13.3 percent. Our earlier article provides insights into the performance and energy improvement of DMP by analyzing pipeline flushes, overhead of predicated instructions, performance impact of different DMP design choices, the confidence estimator's behavior, and the number of fetched and executed instructions.[7]

DMP contributions

DMP provides a novel way to solve the branch-handling problem by dynamically predicating a wide range of hard-to-predict branches at runtime in an energy-efficient way that does not significantly increase the hardware complexity. To enable this solution, the DMP concept makes the following contributions:

- It enables dynamic predication of branches in complex control-flow graphs rather than limiting dynamic predication only to simple hammock branches. It introduces frequently-hammocks. With frequently-hammocks, DMP can eliminate branch mispredictions caused by a much larger set of branches than previous branch-processing techniques can.
- It overcomes the three major limitations of software predication we described earlier, thus offering a comprehensive solution that provides runtime adaptivity to predication and generalizes predication so that it can be applied to a wide range of CFGs without requiring major ISA changes.

- Our comprehensive evaluations of different branch-processing paradigms on both the hardware side (dynamic hammock predication, dual and multipath execution) and the software side (predication and wish branches) show that DMP provides both the best performance and the highest energy-efficiency.[7]

The proposed DMP mechanism still requires some ISA support. A cost-efficient hardware mechanism to detect diverge branches and CFM points at runtime would eliminate the need to change the ISA. Developing such mechanisms is a promising area of future work. The results we presented are based on our initial implementation of DMP using relatively simple compiler and hardware heuristics and algorithms. DMP's effectiveness can be further increased by future research aimed at improving these techniques. On the compiler side, better heuristics and profiling techniques can be developed to select diverge branches and CFM points.[11] On the hardware side, better confidence estimators could critically affect the performance of dynamic predication. Finally, an energy-efficient solution that would reduce the misprediction penalty of branches in nonmerging control flow is a difficult but exciting area of future research. MICRO

Acknowledgments

We thank Chang Joo Lee, Paul Racunas, Veynu Narasiman, Nhon Quach, Derek Chiou, Eric Sprangle, Jared Stark, and the members of the HPS research group. We gratefully acknowledge the support of the Cockrell Foundation, Intel Corporation, and the Advanced Technology Program of the Texas Higher Education Coordinating Board.

References

1. O. Mutlu et al., ''Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors,'' *Proc. 9th Int'l Symp. High-Performance Computer Architecture* (HPCA 03), IEEE CS Press, 2003, pp. 129-140.
2. T.Y. Morad et al., ''Performance, Power Efficiency and Scalability of Asymmetric Cluster Chip Multiprocessors,'' *Computer Architecture Letters*, vol. 4, no. 1, July 2005.
3. J.R. Allen et al., ''Conversion of Control Dependence to Data Dependence,'' *Proc. 10th Ann. Symp. Principles of Programming Languages* (POPL 83), ACM Press, 1983, pp. 177-189.
4. Y. Choi et al., ''The Impact of If-Conversion and Branch Prediction on Program Execution on the Intel Itanium Processor,'' *Proc. 34th Ann. Int'l Symp. Microarchitecture* (Micro 01), IEEE CS Press, 2001, pp. 182-191.
5. H. Kim et al., ''Wish Branches: Combining Conditional Branching and Predication for Adaptive Predicated Execution,'' *Proc. 38th Ann. IEEE/ACM Int'l Symp. Microarchitecture* (Micro 05), IEEE CS Press, 2005, pp. 43-54.
6. A. Klauser et al., ''Dynamic Hammock Predication for Nonpredicated Instruction Set Architectures,'' *Proc. 7th Int'l Conf. Parallel Architectures and Compilation Techniques* (PACT 98), IEEE CS Press, 1998, pp. 278-285.
7. H. Kim et al., ''Diverge-Merge Processor (DMP): Dynamic Predicated Execution of Complex Control-Flow Graphs Based on Frequently Executed Paths,'' *Proc. 39th Ann. IEEE/ACM Int'l Symp. Microarchitecture* (Micro 06), IEEE CS Press, 2006, pp. 53-64.
8. S.A. Mahlke et al., ''Effective Compiler Support for Predicated Execution Using the Hyperblock,'' *Proc. 25th Ann. Int'l Symp. Microarchitecture* (Micro 92), IEEE Press, 1992, pp. 45-54.
9. E. Jacobsen, E. Rotenberg, and J.E. Smith, ''Assigning Confidence to Conditional Branch Predictions,'' *Proc. 29th Ann. IEEE/ACM Int'l Symp. Microarchitecture* (Micro 96), IEEE Press, 1996, pp. 142-152.
10. P.H. Wang et al., ''Register Renaming and Scheduling for Dynamic Execution of Predicated Code,'' *Proc. 7th Int'l Symp. High-Performance Computer Architecture* (HPCA 01), IEEE CS Press, 2001, pp. 15-26.

11. H. Kim et al., ''Profile-Assisted Compiler Support for Dynamic Predication in Diverge-Merge Processors,'' to be published in *Proc. IEEE/ACM Int'l Symp. Code Generation and Optimization* (CGO 07), IEEE CS Press, 2007.
12. T. Heil and J.E. Smith. *Selective Dual Path Execution,* tech. report, Dept. of Electrical and Computer Engineering, Univ. of Wisconsin-Madison, 1996.
13. A. Klauser, A. Paithankar, and D. Grunwald, ''Selective Eager Execution on the PolyPath Architecture,'' *Proc. 25th Ann. Int'l Symp. Computer Architecture* (ISCA 98), IEEE CS Press, 1998, pp. 250-271.
14. S.A. Mahlke et al., ''Characterizing the Impact of Predicated Execution on Branch Prediction,'' *Proc. 27th Ann. Int'l Symp. Microarchitecture* (Micro 94), IEEE Press, 1994, pp. 217-227.
15. A. Seznec, ''Analysis of the O-Geometric History Length Branch Predictor,'' *Proc. 32nd Ann. Int'l Symp. Computer Architecture* (ISCA 05), IEEE CS Press, 2005, pp. 394-405.

Hyesoon Kim is a PhD candidate in the Electrical and Computer Engineering Department of the University of Texas at Austin. Her research interests include high-performance energy-efficient computer architectures and programmer-compiler-architecture interaction. Kim has a BS in mechanical engineering from Korea Advanced Institute of Science and Technology, an MS in mechanical engineering from Seoul National University, and an MS in computer engineering from the University of Texas at Austin. She is a student member of the IEEE and the ACM.

José A. Joao is a PhD student in the Electrical and Computer Engineering Department of the University of Texas at Austin. His research interests include high-performance microarchitectures and compiler-architecture interaction. Joao has an MS in computer engineering from the University of Texas at Austin and a BS in electronics engineering from Universidad Nacional de la Patagonia San Juan Bosco, Argentina, where he is an assistant professor (on leave). He was a recipient of the Microelectronics and Computer Development Fellowship from the University of Texas at Austin during 2003-2005. He is a student member of the IEEE and the ACM.

Onur Mutlu is a researcher at Microsoft Research. His research interests include computer architecture, especially the interactions between software and microarchitecture. Mutlu has BS degrees in computer engineering and psychology from the University of Michigan, Ann Arbor, and PhD and MS degrees in electrical and computer engineering from the University of Texas at Austin. He was a recipient of the Intel PhD fellowship in 2004 and the University of Texas George H. Mitchell Award for Excellence in Graduate Research in 2005. He is a 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 benefits of future process technology to create more effective microarchitectures for future microprocessors. He is coauthor 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.

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