Diverge-Merge Processor (DMP): Dynamic Predicated Execution of Complex Control-Flow Graphs Based on Frequently Executed Paths

Hyesoon Kim José A. Joao Onur Mutlu Yale N. Patt

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

TR-HPS-2006-008
September 2006

This page is intentionally left blank.

Diverge-Merge Processor (DMP): Dynamic Predicated Execution of Complex Control-Flow Graphs Based on Frequently Executed Paths

Hyesoon Kim José A. Joao Onur Mutlu§* Yale N. Patt

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

§Microsoft Research
onur@microsoft.com

Abstract

This paper proposes a new processor architecture for handling hard-to-predict branches, the diverge-merge processor (DMP)*. The goal of this paradigm is to eliminate branch mispredictions due to hard-to-predict dynamic branches by dynamically predicating them without requiring ISA support for predicate registers and predicated instructions. To achieve this without incurring large hardware cost and complexity, the compiler provides control-flow information by hints and the processor dynamically predicates instructions only on frequently executed program paths. The key insight behind DMP is that most control-flow graphs look and behave like simple hammock (if-else) structures when only frequently executed paths in the graphs are considered. Therefore, DMP can dynamically predicate a much larger set of branches than simple hammock branches.*

Our evaluations show that DMP outperforms a baseline processor with an aggressive branch predictor by 19.3% on average over SPEC integer 95 and 2000 benchmarks, through a reduction of 38% in pipeline flushes due to branch mispredictions, while consuming 9.0% less energy. We also compare DMP with previously proposed predication and dual-path/multipath execution paradigms in terms of performance, complexity, and energy consumption, and find that DMP is the highest performance and also the most energy-efficient design.

1. Introduction

State-of-the-art high performance processors employ deep pipelines to extract instruction level parallelism (ILP) and to support high clock frequencies. In the near future, processors are expected to support a large number of in-flight instructions [29, 41, 10, 7, 13] to extract both ILP and memory-level parallelism (MLP). As shown by previous research [26, 39, 40, 29, 41], the performance improvement provided by both pipelining and large instruction windows critically depends on the accuracy of the processor's branch predictor. Branch predictors still remain imperfect despite decades of intensive research in branch prediction. Hard-to-predict branches not only limit processor performance but also result in wasted energy consumption.

Predication has been used to avoid pipeline flushes due to branch mispredictions by converting control dependencies into data dependencies [2]. With predication, the processor fetches instructions from both paths of a branch but commits only results from the correct path, effectively avoiding the pipeline flush associated with a branch misprediction. However, predication has the following problems/limitations:

1. It requires significant support (i.e. predicate registers and predicated instructions) in the instruction set architecture (ISA).
2. Statically predicated code incurs the performance overhead of predicated execution regardless of whether a branch is easy to predict or hard to predict at run-time. The overhead of predicated code is twofolds: (i) the processor always has to fetch instructions from both paths of an if-converted branch, (ii) the processor cannot execute predicated instructions or instructions that are dependent on them until the predicate value is resolved, causing additional delay in execution. Previous research showed that predicated execution sometimes hurts processor performance due to this overhead [9, 21].
3. A large subset of control-flow graphs are usually not converted to predicated code because either the compiler cannot if-convert (i.e. predicate) them or the overhead of predicated execution is high. A control-flow graph that has a function call, a loop, too many exit points, or too many instructions between an entry point and an exit point are examples [2, 31, 27, 9, 43, 30].

Several approaches were proposed to solve these problems/limitations. Dynamic-hammock-predication [22] was proposed to predicate branches without ISA support. However, dynamic-hammock-predication can predicate only simple hammock branches (simple if-else structures

*This work was done while the author was with UT-Austin.

with no nested branches), which account for only a small subset of the mispredicted branches [22]. Wish branches [21] were proposed to reduce the overhead of predicated execution. However, wish branches inherit the limitations of software predication (1 and 3 above) with the exception that they can be applied to loop branches.

Our goal in this paper is to devise a comprehensive technique that overcomes the three problems/limitations of predication so that more processors can employ predicated execution to reduce the misprediction penalty due to hard-to-predict branches.

We propose a new processor architecture, called the *Diverge-Merge Processor (DMP)*. DMP dynamically predicates not only simple but also complex control-flow graphs without requiring predicate registers and predicated instructions in the ISA and without incurring large hardware/energy cost and complexity. The key mechanism of DMP is that it dynamically predicates instructions *only on frequently executed control-flow paths* and *only if a branch is hard-to-predict at run-time*. Dynamically predicating only the frequently executed paths allows DMP to achieve two benefits at the same time: 1) the processor can reduce the overhead of predicated execution since it does not need to fetch/execute *all* instructions that are control-dependent on the predicated branch, 2) the processor can dynamically predicate a large set of control-flow graphs because a complex control-flow graph can look and behave like a simple hammock structure when only frequently executed paths are considered.

Figure 1 shows a control-flow graph example to illustrate the key insight behind DMP. In software predication, if the compiler estimates that the branch at block A is hard-to-predict, it would convert blocks B, C, D, E, F, and G to predicated code and all these blocks would be executed together even though blocks D, F, and G are not frequently executed at run-time [30].[1] In contrast, DMP considers frequently executed paths at run-time, so it can dynamically predicate *only* blocks B, C, and E. To simplify the hardware, DMP uses some control-flow information provided by the compiler. The compiler identifies and marks suitable branches as candidates for dynamic predication. These branches are called *diverge branches*. The compiler also selects a control-flow merge (or reconvergence) point corresponding to each diverge branch. In this example, the compiler marks the branch at block A as a diverge branch and the entry of block H as a control-flow merge (CFM) point. Instead of the compiler specifying which blocks are predicated (and thus fetched), the processor decides what to fetch/predicate at run-time. If a diverge branch is estimated to be low-confidence at run-time, the processor follows and dynamically predicates both paths after the branch until the CFM point. The processor follows the branch predictor outcomes on the two paths to fetch only the frequently executed blocks between a diverge branch and a CFM point.

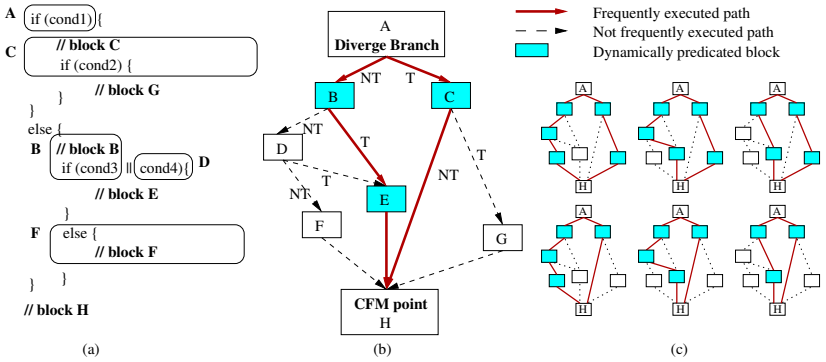

Figure 1. Control-flow graph (CFG) example: (a) source code (b) CFG (c) possible paths (hammocks) that can be predicated by DMP

The compiler could predicate only blocks B, C, and E based on profiling [28] rather than predicating all control-dependent blocks. Unfortunately, frequently executed paths change at run-time (depending on the input data set and program phase), and code predicated for only a

[1] If the compiler does not predicate all basic blocks between A and H because one of the branches is easy-to-predict, then the remaining easy-to-predict branch is likely to become a hard-to-predict branch after if-conversion. This problem is called misprediction migration [3, 38]. Therefore, the compiler (e.g. ORC [30]) usually predicates all control-flow dependent basic blocks inside a region (the region is A,B,C,D,E,F,G and H in this example.). This problem can be mitigated with reverse if-conversion [45, 4] or by incorporating predicate information into the branch history register [3].

few paths can hurt performance if other paths turn out to be frequently executed. In contrast, DMP determines and follows frequently executed paths at run-time and therefore it can flexibly adapt its dynamic predication to run-time changes (Figure 1c shows the possible hammock-shaped paths that can be predicated by DMP for the example control-flow graph). Thus, DMP can dynamically predicate hard-to-predict instances of a branch with less overhead than static predication and with minimal support from the compiler. Furthermore, DMP can predicate a much wider range of control-flow graphs than dynamic-hammock-predication [22] because a control-flow graph does not *have to* be a simple if-else structure to be dynamically predicated; it just needs to *look like* a simple hammock when only frequently executed paths are considered.

Our evaluation shows that DMP improves performance by 19.3% over a baseline processor that uses an aggressive 64KB branch predictor, without significantly increasing maximum power requirements. DMP reduces the number of pipeline flushes by 38%, which results in a 23% reduction in the number of fetched instructions and a 9.0% reduction in dynamic energy consumption. This paper provides a detailed description and analysis of DMP as well as a comparison of its performance, hardware complexity, and power/energy consumption with several previously published branch processing paradigms.

2. The Diverge-Merge Concept

2.1. The Basic Idea

The compiler identifies conditional branches with control flow suitable for dynamic predication as *diverge branches*. A diverge branch is a branch instruction after which the execution of the program *usually* reconverges at a control-independent point in the control-flow graph, a point we call the *control-flow merge (CFM) point*. In other words, diverge branches result in hammock-shaped control flow based on *frequently executed paths in the control-flow graph* of the program but they are not necessarily simple hammock branches that *require* the control-flow graph to be hammock-shaped. The compiler also identifies a CFM point associated with the diverge branch. Diverge branches and CFM points are conveyed to the microarchitecture through modifications in the ISA, which are described in Section 3.13.

When the processor fetches a diverge branch, it estimates whether or not the branch is hard to predict using a branch confidence estimator. If the diverge branch has low confidence, the processor enters *dynamic predication mode (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. If the diverge branch is actually mispredicted, then the processor does not need to flush its pipeline since instructions on both paths of the branch are already fetched and the instructions on the wrong path will become NOPs through dynamic predication.

In this section, we describe the basic concepts of the three major mechanisms to support diverge-merge processing: instruction fetch support, select-μops, and loop branches. A detailed implementation of DMP is described in Section 3.

2.1.1. Instruction Fetch Support In dpred-mode, the processor fetches instructions from both directions (taken and not-taken paths) of a diverge branch using two program counter (PC) registers and a round-robin scheme to fetch from the two paths in alternate cycles. On each path, the processor follows the outcomes of the branch predictor. Note that the outcomes of the branch predictor favor the frequently executed basic blocks in the control flow graph. The processor uses a separate global branch history register (GHR) to predict the next fetch address on each path, and it checks whether the predicted next fetch address is the CFM point of the diverge branch.[2] If the processor reaches the CFM point on one path, it stops fetching from that path and fetches from only the other path. When the processor reaches the CFM point on both paths, it exits dpred-mode.

[2] When the predicted next fetch address is the CFM point of the diverge branch, the processor considers that it has reached the CFM point.

2.1.2. Select-μops Instructions after the CFM point should have data dependencies on instructions from only the correct path of a diverge branch. Before the diverge branch is executed, the processor does not know which path is correct. Instead of waiting for the resolution of the diverge branch, the processor inserts select-μops to continue renaming/execution after exiting dpred-mode. Select-μops are similar to the ϕ-functions in the static single-assignment (SSA) form [14] in that they "merge" the register values produced on both sides of the hammock.[3] Select-μops ensure that instructions dependent on the register values produced on either side of the hammock are supplied with the correct data values that depend on the correct direction of the diverge branch. 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 (i.e. depends on) the output of a select-μop. Such an instruction will be executed after the diverge branch is resolved. However, instructions that are not dependent on select-μops are executed as soon as their sources are ready without waiting for the resolution of the diverge branch. Figure 2 illustrates the dynamic predication process. Note that instructions in blocks C, B, and E, which are fetched during dpred-mode, are also executed before the resolution of the diverge branch.

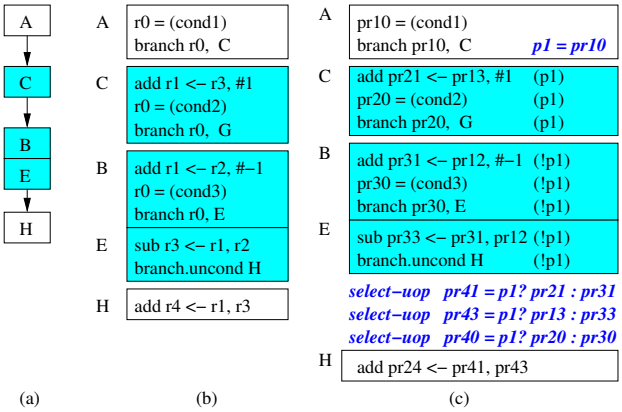

Figure 2. An example of how the instruction stream in Figure 1b is dynamically predicated: (a) fetched blocks (b) fetched assembly instructions (c) instructions after register renaming

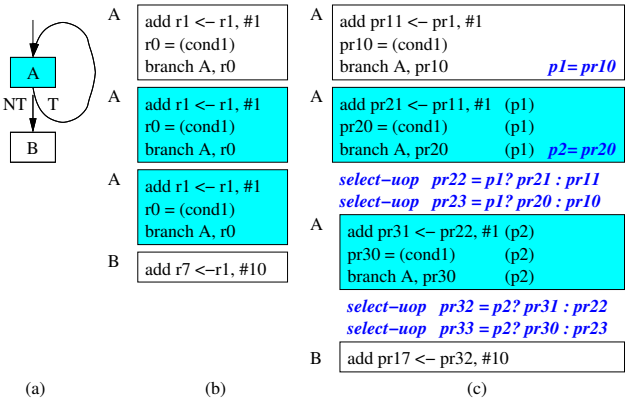

Figure 3. An example of how a loop-type diverge branch is dynamically predicated: (a) CFG (b) fetched assembly instructions (c) instructions after register renaming

2.1.3. Loop Branches DMP can dynamically predicate loop branches. The benefit of dynamically predicating loop branches using DMP is very similar to the benefit of *wish loops* [21]. The key mechanism to predicate a loop-type diverge branch is that the processor needs to predicate each loop iteration separately. This is accomplished by using a different predicate register for each iteration and inserting select-μops

[3] Select-μops handle the merging of only register values. We explain how memory values are handled in Section 3.10.

after each iteration. Select-μops choose between live-out register values before and after the execution of a loop iteration, based on the outcome of each dynamic instance of the loop branch. Instructions that are executed in later iterations and that are dependent on live-outs of previous predicated iterations source the outputs of select-μops. Similarly, instructions that are fetched after the processor exits the loop and that are dependent on registers produced within the loop source the outputs of select-μops so that they receive the correct source values even though the loop branch may be mispredicted. The pipeline does not need to be flushed if a predicated loop is iterated more times than it should be because the predicated instructions in the extra loop iterations will become NOPs and the live-out values from the correct last iteration will be propagated to dependent instructions via select-μops. Figure 3 illustrates the dynamic predication process of a loop-type diverge branch (The processor enters dpred-mode after the first iteration and exits after the third iteration).

There is a negative effect of predicating loops: instructions that source the results of a previous loop iteration (i.e. loop-carried dependencies) cannot be executed until the loop-type diverge branch is resolved because such instructions are dependent on select-μops. However, we found that the negative effect of this execution delay is much less than the benefit of reducing pipeline flushes due to loop branch mispredictions. Note that the dynamic predication of a loop does not provide any performance benefit if the branch predictor iterates the loop fewer times than required by correct execution, or if the predictor has not exited the loop by the time the loop branch is resolved.

2.2. DMP vs. Other Branch Processing Paradigms

We compare DMP with five previously proposed mechanisms in predication and multipath execution paradigms: dynamic-hammock-predication [22], software predication [2, 31], wish branches [21], selective/limited dual-path execution (dual-path) [18, 15], and multi-path/PolyPath execution (multipath) [33, 24]. First, we classify control-flow graphs (CFGs) into five different categories to illustrate the differences between these mechanisms more clearly.

Figure 4 shows examples of the five different CFG types. Simple hammock (Figure 4a) is an `if` or `if-else` structure that does not have any nested branches inside the hammock. Nested hammock (Figure 4b) is an `if-else` structure that has multiple levels of nested branches. Frequently-hammock (Figure 4c) is a CFG that becomes a simple hammock if we consider only frequently executed paths. Loop (Figure 4d) is a cyclic CFG (`for`, `do-while`, or `while` structure). Non-merging control-flow (Figure 4e) is a CFG that does not have a control-flow merge point even if we consider only frequently executed paths.[4] Figure 5 shows the frequency of branch mispredictions due to each CFG type. Table 1 summarizes which blocks are fetched/predicated in different processing models for each CFG type, assuming that the branch in block A is hard to predict.

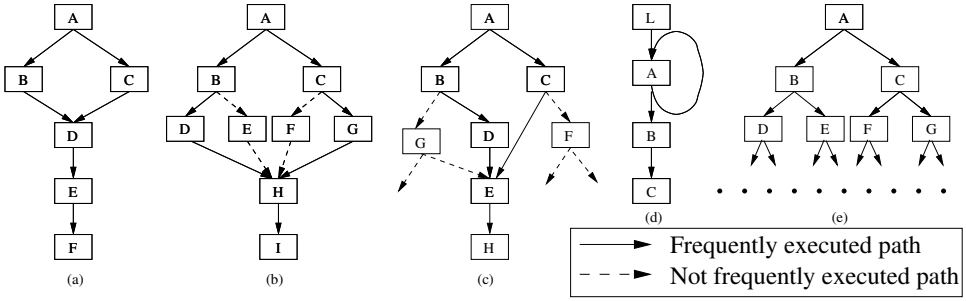

Figure 4. Control-flow graphs: (a) simple hammock (b) nested hammock (c) frequently-hammock (d) loop (e) non-merging control flow

Dynamic-hammock-predication can predicate only simple hammocks which account for 12% of all mispredicted branches. Simple hammocks by themselves account for a significant percentage of mispredictions in only two benchmarks: vpr (40%) and twolf (36%). We expect

[4] If the number of static instructions between a branch and the closest control-flow merge point exceeds a certain number (T), we consider that the CFG does not have a control-flow merge point. T=200 in our experiments.

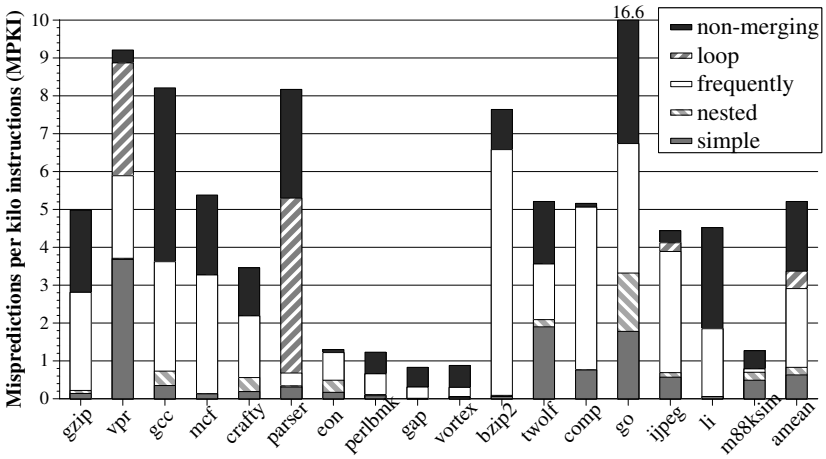

Figure 5. Distribution of mispredicted branches based on CFG type

Table 1. Fetched instructions in different processing models (after the branch at A is estimated to be low-confidence) We assume that the loop branch in block A (Figure 4d) is predicted taken twice after it is estimated to be low-confidence.

Processing model	simple hammock	nested hammock	frequently-hammock	loop	non-merging
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 don't/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 don't/can't predicate	A, A, B, C	can't predicate
Dual-path	path1: B, D, E, F path2: C, D, E, F	path1: B, D, H, I path2: C, G, H, I	path1: B, D, E, H path2: C, E, H	path1: A, A, B, C path2: B, C	path1: B ... path2: C ...

dynamic-hammock-predication will improve the performance of these two benchmarks.

Software predication can predicate both simple and nested hammocks, which in total account for 16% of all mispredicted branches. Software predication fetches all basic blocks between an if-converted branch and the corresponding control-flow merge point. For example, in the nested hammock case (Figure 4b), 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 do not predicate frequently-hammocks since the overhead of predicated code would be too high if these CFGs include function calls, cyclic control-flow, too many exit points, or too many instructions [2, 31, 43, 27, 9, 30]. Note that hyperblock formation [28] 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 run-time. Even if we assume that software predication can predicate all frequently-hammocks, it could predicate up to 56% of all mispredicted branches.

Wish branches can predicate even loops, which account for 10% of all mispredicted branches, in addition to what software predication can do. The main difference between wish branches and software predication is that the wish branch mechanism can selectively predicate each dynamic instance of a branch. With wish branches, a branch is predicated only if it is hard to predict at run-time, whereas with software predication a branch is predicated 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/predicated. Therefore, wish branches also have higher performance overhead for nested hammocks than DMP.

Note that software predication (and wish branches) can eliminate a branch misprediction due to a branch that is control-dependent on another hard-to-predict branch (e.g. branch at B is control-dependent on branch at A in Figure 4b), since it predicates all the basic blocks within a nested hammock. This benefit is not possible with any of the other paradigms except multipath, but we found that it provides significant performance benefit only in two benchmarks (3% in twolf, 2% in go).

Selective/limited dual-path execution fetches from two paths after a hard-to-predict branch. The 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 does not have to reconverge. Hence, dual-path can potentially eliminate the branch misprediction penalty for all five CFG types. However, the dual-path mechanism needs to fetch a larger number of instructions than any of the other mechanisms (except multipath) because it continues fetching from two paths until the hard-to-predict branch is resolved even though the processor may have already reached a control-independent point in the CFG. For example, in the simple hammock case (Figure 4a), DMP fetches blocks D, E, and F only once, but dual-path fetches D, E, and F twice (once for each path). Therefore, the overhead of dual-path is much higher than that of DMP. Detailed comparisons of the overhead and performance of different processing models are provided in Section 5.

Multipath execution is a generalized form of dual-path execution in that it fetches both paths after *every* low-confidence branch and therefore it can execute along many (more than two) different paths at the same time. This increases the probability of having the correct path in the processor's instruction window. However, only one of the outstanding paths is the correct path and instructions on every other path have to be flushed. Furthermore, instructions after a control-flow independent point have to be fetched/executed separately for each path (like dual-path but unlike DMP), which causes the processing resources to be wasted for instructions on all paths but one. For example, if the number of outstanding paths is 8, then a multipath processor wastes 87.5% of its fetch/execution resources for wrong-path/useless instructions even after a control-independent point. Hence, the overhead of multipath is much higher than that of DMP. In the example of Table 1 the behavior of multipath is the same as that of dual-path because the example assumes there is only one hard-to-predict branch to simplify the explanation.

DMP can predicate simple hammocks, nested hammocks, frequently-hammocks, and loops. On average, these four CFG types account for 66% of all branch mispredictions. The number of fetched instructions in DMP is less than or equal to other mechanisms for all CFG types, as shown in Table 1. Hence, we expect DMP to eliminate branch mispredictions more efficiently (i.e. with less overhead) than the other processing paradigms.

3. Implementation of DMP

3.1. Entering Dynamic Predication Mode

The diverge-merge processor enters dynamic predication mode (dpred-mode) if a diverge branch is estimated to be low-confidence at run-time. When the processor enters dpred-mode, it needs to do the following:

1. The front-end stores the address of the CFM point associated with the diverge branch into a buffer called CFM register. The processor also marks the diverge branch as the branch that caused entry into dpred-mode.
2. The front-end forks (i.e. creates a copy of) the return address stack (RAS) and the GHR when the processor enters dpred-mode. In dpred-mode, the processor accesses the same branch predictor table with two different GHRs (one for each path) but only correct path instructions update the table after they commit. A separate RAS is needed for each path. The processor forks the register alias table (RAT) when the diverge branch is renamed so that each path uses a separate RAT for register renaming in dpred-mode. This hardware support is similar to the dual-path execution mechanisms [1].
3. The front-end allocates a predicate register for the initiated dpred-mode. An instruction fetched in dpred-mode carries the predicate register identifier (id) with an extra bit indicating whether the instruction is on the taken or the not-taken path of the diverge branch.

3.2. Short Hammocks

Frequently-mispredicted hammock branches with few instructions before the CFM point are good candidates to be *always* predicated, even if the confidence on the branch prediction is high. The reason for this heuristic is that while the cost of mispredicting a short-hammock branch is high (flushing mostly control-independent instructions that were fetched after the CFM point), the cost of dynamic predication of a

short-hammock branch is low (useless execution of just the few instructions on the wrong-path of the branch). Therefore, always predicating short-hammock diverge branch candidates with very low dynamic predication cost is a reasonable trade-off.

3.3. Multiple CFM points

DMP can support more than one CFM point for a diverge branch to enable the predication of dynamic hammocks that start from the same branch but end at different control-independent points. The compiler provides multiple CFM points. At run-time, the processor chooses the CFM point reached first on any path of the diverge branch and uses it to end dpred-mode. To support multiple CFM points, the CFM register is extended to hold multiple CFM-point addresses.

3.4. Return CFM points

Some function calls are ended by different return instructions on the taken and not-taken paths of a diverge branch. In this case, the CFM point is the instruction executed after the return, whose address is not known at compile time because it depends on the caller position. We introduce a special type of CFM point called *return CFM* to handle this case. When a diverge branch includes a return CFM, the processor does not look for a particular CFM point address to end dpred-mode, but for the execution of a return instruction.

3.5. Exiting Dynamic Predication Mode

DMP exits dpred-mode when either (1) both paths of a diverge branch have reached the corresponding CFM point or (2) a diverge branch is resolved. The processor marks the last instruction fetched in dpred-mode (i.e. the last predicated instruction). The last predicated instruction triggers the insertion of select-μops after it is renamed.

DMP employs two policies to exit dpred-mode early to increase the benefit and reduce the overhead of dynamic predication:

1. **Counter Policy:** CFM points are chosen based on frequently executed paths determined through compile-time profiling. At run-time, the processor might not reach a CFM point if the branch predictor predicts that a different path should be executed. For example, in Figure 4c, the processor could fetch blocks C and F. In that case, the processor never reaches the CFM point and hence continuing dynamic predication is less likely to provide benefit. To stop dynamic predication early (before the diverge branch is resolved) in such cases, we use a heuristic. If the processor does not reach the CFM point until a certain number of instructions (N) are fetched on any of the two paths, it exits dpred-mode. N can be a single global threshold or it can be chosen by the compiler for each diverge branch. We found that a per-branch threshold provides 2.3% higher performance than a global threshold because the number of instructions executed to reach the CFM point varies across diverge branches. After exiting dpred-mode early, the processor continues to fetch from only the predicted direction of the diverge branch.

2. **Yield Policy:** DMP fetches only two paths at the same time. If the processor encounters another low-confidence diverge branch during dpred-mode, it has two choices: it either treats the branch as a normal (non-diverge) branch or exits dpred-mode for the earlier diverge branch and enters dpred-mode for the later branch. We found that a low-confidence diverge branch seen on the predicted path of a dpred-mode-causing diverge branch usually has a higher probability to be mispredicted than the dpred-mode-causing diverge branch. Moreover, dynamically predicating the later control-flow dependent diverge branch usually has less overhead than predicating the earlier diverge branch because the number of instructions inside the CFG of the later branch is smaller (since the later branch is usually a nested branch of the previous diverge branch). Therefore, our DMP implementation exits dpred-mode for the earlier diverge branch and enters dpred-mode for the later diverge branch.

3.6. Select-μop Mechanism

Select-μops are inserted when the processor reaches the CFM point on both paths. Select-μops choose data values that were produced from the two paths of a diverge branch so that instructions after the CFM point receive correct data values from select-μops. Our select-μop

generation mechanism is similar to Wang et al.'s [44]. However, our scheme is simpler than theirs because it needs to compare only two RATs to generate the select-μops. A possible implementation of our scheme is explained below.

When a diverge branch that caused entry into dpred-mode reaches the renaming stage, the processor forks the RAT. The processor uses two different RATs, one for each path of the diverge branch. We extend the RAT with one extra bit (M -modified-) per entry to indicate that the corresponding architectural register has been renamed in dpred-mode. Upon entering dpred-mode, all M bits are cleared. When an architectural register is renamed in dpred-mode, its M bit is set.

When the last predicated instruction reaches the register renaming stage, the select-μop insertion logic compares the two RATs.[5] If the M bit is set for an architectural register in either of the two RATs, a select-μop is inserted to choose, according to the predicate register value, between the two physical registers assigned to that architectural register in the two RATs. A select-μop allocates a new physical register ($\texttt{PR}_{new}$) for the architectural register. Conceptually, the operation of a select-μop can be summarized as $\texttt{PR}_{new}\texttt{=(predicate\_register\_value)?PR}_T\texttt{:PR}_{NT}$, where $\texttt{PR}_T\texttt{(PR}_{NT}\texttt{)}$ is the physical register assigned to the architectural register in the RAT of the taken (not-taken) path.

A select-μop is executed when the predicate value and the selected source operand are ready. As a performance optimization, a select-μop does not wait for a source register that will not be selected. Note that the select-μop generation logic operates in parallel with work done in other pipeline stages and its implementation does not increase the pipeline depth of the processor.

3.7. Handling Loop Branches

Loop branches are treated differently from non-loop branches. One direction of a loop branch is the exit of the loop and the other direction is one more iteration of the loop. When the processor enters dpred-mode for a loop branch, only one path (the loop iteration direction) is executed and the processor will fetch the same static loop branch again. Entering dpred-mode for a loop branch always implies the execution of one more loop iteration.

The processor enters dpred-mode for a loop if the loop-type diverge branch is low confidence. When the processor fetches the same static loop branch again during dpred-mode, it exits dpred-mode and inserts select-μops. If the branch is predicted to iterate the loop once more, the processor enters dpred-mode again with a different predicate register id[6], regardless of the confidence of the branch prediction. In other words, once the processor dynamically predicates one iteration of the loop, it continues to dynamically predicate the iterations until the loop is exited by the branch predictor. The processor stores the predicate register ids associated with the same static loop branch in a small buffer and these are later used when the branch is resolved as we will describe in Section 3.8. If the branch is predicted to exit the loop, the processor does not enter dpred-mode again but it starts to fetch from the exit of the loop after inserting select-μops.

3.8. Resolution of Diverge Branches

When a diverge branch that caused entry into dpred-mode is resolved, the processor does the following:

1. It broadcasts the predicate register id of the diverge branch with the correct branch direction (taken or not-taken). Instructions with the same predicate id and the same direction are said to be predicated-TRUE and those with the same predicate id but different direction are said to be predicated-FALSE.
2. If the processor is still in dpred-mode for that predicate register id, it simply exits dpred-mode and continues fetching only from the correct path as determined by the resolved branch. If the processor has already exited dpred-mode, it does not need to take any special action. In either case, the pipeline is not flushed.

[5] This comparison is actually performed incrementally every time a register is renamed in dpred-mode so that no extra cycles are wasted for select-μop generation. We simplify the explanation by describing it as if it happens at once at the end of dpred-mode.

[6] DMP has a limited number of predicate registers (32 in our model). Note that these registers are not architecturally visible.

3. If a loop-type diverge branch exits the loop (i.e. resolved as not-taken in a backward loop), the processor also broadcasts the predicate id's that were assigned for later loop iterations along with the correct branch direction in consecutive cycles.[7] This ensures that the select-μops after each later loop iteration choose the correct live-out values.

DMP flushes its pipeline for any mispredicted branch that did not cause entry into dpred-mode, such as a mispredicted branch that was fetched in dpred-mode and turned out to be predicated-TRUE.

3.9. Instruction Execution and Retirement

Dynamically predicated instructions are executed just like other instructions (except for store-load forwarding described in Section 3.10). Since these instructions depend on the predicate value only for retirement purposes, they can be executed before the predicate value (i.e. the diverge branch) is resolved. If the predicate value is known to be FALSE, the processor does not need to execute the instructions or allocate resources for them. Nonetheless, all predicated instructions consume retirement bandwidth. When a predicated-FALSE instruction is ready to be retired, the processor simply frees the physical register (along with other resources) allocated for that instruction and does not update the architectural state with its results.[8] The predicate register associated with dpred-mode is released when the last predicated instruction is retired.

3.10. Load and Store Instructions

Dynamically predicated load instructions are executed like normal load instructions. Dynamically predicated store instructions are sent to the store buffer with their predicate register id. However, a predicated store instruction is not sent further down the memory system (i.e. into the caches) until it is known to be predicated-TRUE. The processor drops all predicated-FALSE store requests. Thus, DMP 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. The forwarding logic should check not only the addresses but also the predicate register ids. The logic can forward from: (1) a non-predicated store to any later load, (2) a predicated store whose predicate register value is known to be TRUE to any later load, or (3) a predicated store whose predicate register is not ready to a later load with the same predicate register id (i.e. on the same dynamically predicated path). If forwarding is not possible, the load waits. Note that this mechanism and structures to support it are the same as the store-load forwarding mechanism in dynamic-hammock-predication [22]. An out-of-order execution processor that implements software predication or wish branches also requires the same support in the store buffer and store-load forwarding logic.

3.11. Interrupts and Exceptions

DMP does not require any special support for handling interrupts or exceptions. When the pipeline is flushed before servicing the interrupt or exception, any speculative state, including DMP-specific state is also flushed. There is no need to save and restore predicate registers, unlike software predication. The processor restarts in normal mode right after the last architectural retired instruction after coming back from the interrupt/exception service. Exceptions generated by predicated-FALSE instructions are simply dropped.

3.12. Hardware Complexity Analysis

DMP increases hardware complexity compared to current processors but it is an energy efficient design as we will show in Section 5.5. Some of the hardware required for DMP is already present in current processors. For example, select-μops are similar to CMOV operations and complex μop generation and insertion schemes are already implemented in x86 processors. Table 2 summarizes the additional hardware

[7] Note that only one predicate id needs to be broadcast per cycle because select-μops from a later iteration cannot anyway be executed before the select-μops from the previous iteration are executed (since select-μops of the later iteration are dependent on the select-μops of the previous iteration).

[8] In a current out-of-order processor, when an instruction is ready to be retired, the processor frees the physical register allocated by the previous instruction that wrote to the same architectural register. This is exactly how physical registers are freed in DMP for non-predicated and predicated-TRUE instructions. The only difference is that a predicated-FALSE instruction frees the physical register allocated by itself (since that physical register will not be part of the architectural state) rather than the physical register allocated by the previous instruction that wrote to the same architectural register.

support required for DMP and the other processing models. DMP requires slightly more hardware support than dynamic-hammock-predication and dual-path but much less than multipath.

Table 2. Hardware support required for different branch processing paradigms. (m+1) is the maximum number of outstanding paths in multipath.

Hardware	DMP	Dynamic-hammock	Dual-path/Multipath	Software predication	Wish branches
Fetch support	CFM registers, +1 PC round-robin fetch	fetch both paths in simple hammock	+1/m PC round-robin fetch	-	selection between branch/predicated code
Hardware-generated predicate/path IDs	required	required	required (path IDs)	-	-
Branch pred. support	+1 GHR, +1 RAS	-	+1/m GHR, +1/m RAS	-	-
BTB support	mark diverge br./CFM	mark hammock br.	-	-	mark wish branches
Confidence estimator	required	optional (performance)	required	-	required
Decode support	CFM point info	-	-	predicated instructions	predicated instructions
Rename support	+1 RAT	+1 RAT	+1/m RAT	-	
Predicate registers	required	required	-	required	required
Select-μop generation	required	required	-	optional (performance)	optional (performance)
LD-ST forwarding	check predicate	check predicate	check path IDs	check predicate	check predicate
Branch resolution	check flush/no flush predicate id broadcast	check flush/no flush	check flush/no flush	-	check flush/no flush
Retirement	check predicate	check predicate	selective flush	check predicate	check predicate

3.13. ISA Support for Diverge Branches

We present an example of how the compiler can transfer diverge branch and CFM point information to the hardware through simple modifications in the ISA. Diverge branches are distinguished with two bits in the ISA's branch instruction format. The first bit indicates whether or not the branch is a diverge branch and the second bit indicates whether or not a branch is of loop-type. If a branch is a diverge branch, the following *N* bits in the program code are interpreted as the encoding for the associated CFM points. A CFM point address can be encoded as a relative address from the diverge branch address or as an absolute address without the most significant bits. Since CFM points are located close to a diverge branch we found that 10 bits are enough to encode each CFM point selected by our compiler algorithm. The ISA could dedicate a fixed number of bytes to encode CFM points or the number of bytes can vary depending on the number of CFM points for each diverge branch. We allow maximum 3 CFM points per diverge branch. To support early exit (Section 3.5), the compiler also uses *L* extra bits to encode the maximum distance between a branch and its CFM point (*L* is a scaled 4-bit value in our implementation).

4. Methodology

4.1. Simulation Methodology

We use an execution-driven simulator of a processor that implements the Alpha ISA. An aggressive, 64KB branch predictor is used in the baseline processor. The parameters of the baseline processor are shown in Table 3.

Table 3. Baseline processor configuration

Front End	64KB, 2-way, 2-cycle I-cache; fetches up to 3 conditional branches but fetch ends at the first predicted-taken branch; 8 RAT ports
Branch Predictors	64KB (64-bit history, 1021-entry) perceptron branch predictor [20]; 4K-entry BTB 64-entry return address stack; minimum branch misprediction penalty is 30 cycles
Execution Core	8-wide fetch/issue/execute/retire; 512-entry reorder buffer; 128-entry load-store queue; 512 physical registers scheduling window is partitioned into 8 sub-windows of 64 entries each; 4-cycle pipelined wake-up and selection logic
On-chip Caches	L1 D-cache: 64KB, 4-way, 2-cycle, 2 ld/st ports; L2 cache: 1MB, 8-way, 8 banks, 10-cycle, 1 port; LRU replacement and 64B line size
Buses and Memory	300-cycle minimum memory latency; 32 banks; 32B-wide core-to-memory bus at 4:1 frequency ratio; bus latency: 40-cycle round-trip
Prefetcher	Stream prefetcher with 32 streams and 16 cache line prefetch distance (lookahead) [42]
DMP Support	2KB (12-bit history, threshold 14) enhanced JRS confidence estimator [19, 17]; 32 predicate registers; 3 CFM registers (also see Table 2)

We also model a less aggressive (base2) processor to evaluate the DMP concept in a configuration similar to today's processors. Table 4 shows the parameters of the less aggressive processor that are different from the baseline processor.

The experiments are run using the 12 SPEC CPU 2000 integer benchmarks and 5 SPEC 95 integer benchmarks.[9] Table 5 shows the

[9]Gcc, vortex, and perl in SPEC 95 are not included because later versions of these benchmarks are included in SPEC CPU 2000.

Table 4. Less aggressive baseline processor (base2) configuration

Front End	Fetches up to 2 conditional branches but fetch ends at the first predicted-taken branch; 4 RAT ports
Branch Predictors	16KB (31-bit history, 511-entry) perceptron branch predictor [20]; 1K-entry BTB 32-entry return address stack; minimum branch misprediction penalty is 20 cycles
Execution Core	4-wide fetch/issue/execute/retire; 128-entry reorder buffer; 64-entry scheduling window; 48-entry load-store queue 128 physical registers; 3-cycle pipelined wake-up and selection logic
Buses and Memory	200-cycle minimum memory latency; bus latency: 20-cycle round-trip

characteristics of the benchmarks on the baseline processor. All binaries are compiled for the Alpha ISA with the -fast optimizations. We use a binary instrumentation tool that marks diverge branches and their respective CFM points after profiling. The benchmarks are run to completion with a reduced input set [25] to reduce simulation time. In all the IPC (retired Instructions Per Cycle) performance results shown in the rest of the paper for DMP, instructions whose predicate values are FALSE and select-μops inserted to support dynamic predication do not contribute to the instruction count.

Table 5. Characteristics of the benchmarks: baseline IPC, potential IPC improvement with perfect branch prediction (PBP IPC Δ), total number of retired instructions (Insts), number of static diverge branches (Diverge Br.), number of all static branches (All br.), increase in code size with diverge branch and CFM information (Code size Δ), base2 processor IPC (IPC base2), potential IPC improvement with perfect branch prediction on the base2 processor (PBP IPC Δ base2). perl, comp, m88 are the abbreviations for perlbmk, compress, and m88ksim respectively.

	gzip	vpr	gcc	mcf	crafty	parser	eon	perl	gap	vortex	bzip2	twolf	comp	go	ijpeg	li	m88
Base IPC	2.02	1.50	1.25	0.45	2.54	1.50	3.26	2.27	2.88	3.37	1.48	2.18	2.18	0.97	2.73	2.15	3.27
PBP IPC Δ	90%	229%	96%	113%	60%	137%	21%	15%	15%	16%	94%	112%	139%	227%	93%	60%	24%
Insts (M)	249	76	83	111	190	255	129	99	404	284	316	101	150	137	346	248	145
Diverge br.	84	434	1245	62	192	37	116	92	79	250	74	235	16	117	48	18	158
All br. (K)	1.6	4.2	29.5	1.4	5.1	3.7	4.9	9.4	4.6	13	1.4	4.7	0.6	7.7	2	1.2	1.7
Code size Δ(%)	0.12	0.35	0.23	0.1	0.13	0.03	0.01	0.03	0.03	0.09	0.11	0.16	0.02	0.08	0.04	0.02	0.13
IPC base2	1.77	1.39	0.98	0.52	1.76	1.36	2.05	1.36	2.03	1.73	1.39	1.71	1.79	0.86	2.05	1.69	2.10
PBP IPC Δ base2	39%	84%	46%	58%	27%	65%	9%	7%	9%	8%	46%	46%	50%	101%	37%	34%	12%

4.2. Modeling of Other Branch Processing Paradigms

4.2.1. Dynamic-Hammock-Predication Klauser et al. [22] discussed several design configurations for dynamic-hammock-predication. We chose the following design configurations that provide the best performance: (1) Simple hammock branches are marked by the compiler through profiling, (2) A confidence estimator is used to decide when to predicate a simple hammock.

4.2.2. Dual-path Several design choices for dual-path processors were proposed [18, 15, 24, 1]. The dual-path processor we model fetches instructions from two paths of a low confidence branch using a round-robin scheme. To give priority to the predicted path (since the branch predictor is more likely to predict a correct direction), the processor fetches twice as many instructions from the predicted path as from the other path [1]. This is accomplished by fetching from the other path every third cycle. The configuration of the confidence estimator is optimized to maximize the benefit of dual-path (13-bit history, threshold 4). Most of the previous evaluations of dual-path processors increased the fetch/rename/execution bandwidth to support two paths. However, in our model, the baseline, dynamic-hammock-predication, dual-path, multipath, and DMP have the same amount of fetch/rename/execution bandwidth in order to provide fair comparisons.

4.2.3. Multipath The modeled multipath processor starts fetching from both paths *every time* it encounters a low-confidence branch, similarly to PolyPath [24]. The maximum number of outstanding paths is 8, which we found to perform best among 4, 6, 8, 16, or 32 outstanding paths. The processor fetches instructions from each outstanding path using a round-robin scheme.

4.2.4. Limited Software Predication Since the Alpha ISA does not support full predication, we model limited software predication [10] with the following modifications in the DMP mechanism: (1) a diverge branch is always (i.e. statically) converted into predicated code and eliminated from the program, (2) only simple and nested hammocks are converted into predicate code, (3) all basic blocks (instructions) between

[10] we call it limited software predication, because our software predication does not model compiler's optimization effect on if-conversion

a diverge branch and the CFM point of the branch are fetched/predicated, (4) there is no branch misprediction between the diverge branch and the CFM point since all blocks are predicated, (5) a select-$u$op mechanism [44] (similarly to DMP) is employed so that predicated instructions can be executed before the predicate value is ready.

4.2.5. Wish Branches We model wish branches similarly to limited software predication except that: (1) the processor decides whether or not to predicate based on the confidence of branch prediction (same as in DMP), (2) the processor can predicate not only simple and nested hammocks but also loop branches, (3) a wish branch is not eliminated from the program.

4.3. Power Model

We incorporated the Wattch infrastructure [5] into our cycle-accurate simulator. The power model is based on 100nm technology. The frequency we assume is 4GHz for the baseline processor and 1.5GHz for the less aggressive processor. We use the aggressive CC3 clock-gating model in Wattch: unused units dissipate only 10% of their maximum power when they are not accessed [5]. All additional structures and instructions required by DMP are faithfully accounted for in the power model: the confidence estimator, one more RAT/RAS/GHR, select-μop generation/execution logic, additional microcode fields to support select-μops, additional fields in the BTB to mark diverge branches and to cache CFM points, predicate and CFM registers, and modifications to handle load-store forwarding and instruction retirement. Forking of tables and insertion of select-μops are modeled by increasing the dynamic access counters for every relevant structure.

4.4. Compiler Support for Diverge Branch and CFM Point Selection

Diverge branch and CFM point candidates are determined based on a combination of CFG analysis and profiling. Simple hammocks, nested hammocks, and loops are found by the compiler using CFG analysis. To determine frequently-hammocks, the compiler finds CFM point candidates (i.e. post-dominators) considering the portions of a program's control-flow graph that are executed during the profiling run. A branch in a suitable CFG is marked as a possible diverge branch if it is responsible for at least 0.1% of the total number of mispredictions during profiling. A CFM point candidate is selected as a CFM point if it is reached from a diverge branch for at least 30% of the dynamic instances of the branch during the profiling run and if it is within 120 static instructions from the diverge branch. The thresholds used in compiler heuristics are determined experimentally. We used the *train* input sets to collect profiling information.

5. Results

5.1. Performance of the Diverge-Merge Processor

Figure 6 shows the performance improvement of dynamic-hammock-predication, dual-path, multipath, and DMP over the baseline processor. The average IPC improvement over all benchmarks is 3.5% for dynamic-hammock-predication, 4.8% for dual-path, 8.8% for multipath,[11] and 19.3% for DMP. DMP improves the IPC by more than 20% on vpr (58%), mcf (47%), parser (26%), twolf (31%), compress (23%), and ijpeg (25%). A significant portion (more than 60%) of branch mispredictions in these benchmarks is due to branches that can be dynamically predicated by DMP as was shown in Figure 5. Mcf shows additional performance benefit due to the prefetching effect caused by predicated-FALSE instructions. In bzip2, even though 87% of mispredictions are due to frequently-hammocks, DMP improves IPC by only 12.2% over the baseline. Most frequently-hammocks in bzip2 have more than one CFM point and the run-time heuristic used by DMP to decide which CFM point to use for dynamic predication (Section 3.3) does not work well for bzip2.

Dynamic-hammock-predication provides over 10% performance improvement on vpr and twolf because a relatively large portion of mispredictions is due to simple hammocks. The performance benefit of dual-path is higher than that of dynamic-hammock-predication but much

[11] Klauser et al. [22] reported average 5% performance improvement for dynamic-hammock-predication, Farrens et al. [15] reported average 7% performance improvement for dual-path (with extra execution resources to support dual-path), and Klauser and Grunwald [23] reported average 9.3% performance improvement for PolyPath (multipath) with a round-robin fetch scheme. The differences between their and our results are due to different branch predictors, machine configurations, and benchmarks. Our baseline branch predictor is much more accurate than those in previous work.

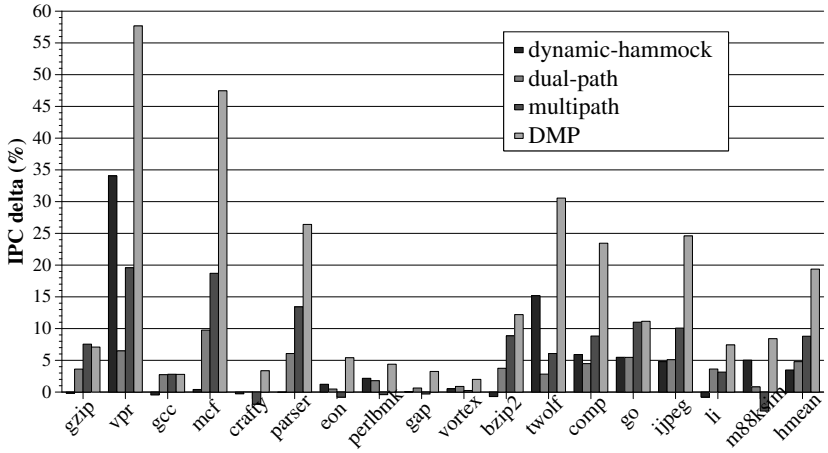

Figure 6. Performance improvement provided by DMP vs. dynamic-hammock-predication, dual-path, and multipath execution

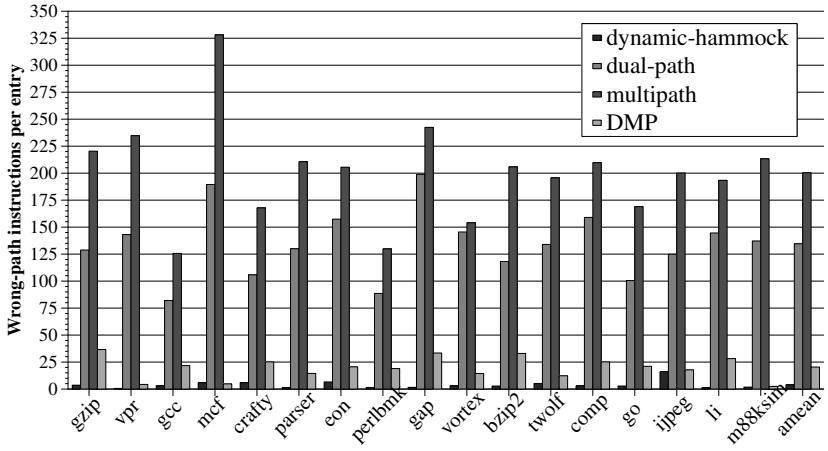

Figure 7. Fetched wrong-path instructions per entry into dynamic-predication/dual-path mode (i.e. per low-confidence branch)

less than that of DMP, even though dual-path is applicable to any kind of CFG. This is due to two reasons. First, dual-path fetches a larger number of instructions from the wrong path compared to dynamic-hammock-predication and DMP, as was shown in Table 1. Figure 7 shows the average number of fetched wrong-path instructions per each entry into dynamic-predication/dual-path mode in the different processors. On average, dual-path fetches 134 wrong-path instructions, which is much higher than 4 for dynamic-hammock-predication, and 20 for DMP (note that this overhead is incurred even if the low-confidence branch turns out to be correctly predicted). Second, dual-path is applicable to one low-confidence branch at a time. While a dual-path processor is fetching from two paths, it cannot perform dual-path execution for another low-confidence branch. However, DMP can diverge again if another low confidence diverge branch is encountered after the processor has reached the CFM point of a previous diverge branch and exited dpred-mode. For this reason, we found that dual-path cannot reduce as many pipeline flushes due to branch mispredictions as DMP. As Figure 8 shows, dual-path reduces pipeline flushes by 18% whereas DMP reduces them by 38%.

Multipath performs better than or similarly to DMP on gzip, gcc, and go. In these benchmarks more than 40% of branch mispredictions are due to non-merging control flow that cannot be predicated by DMP but can be eliminated by multipath. Multipath also performs better than dual-path execution on average because it is applicable to multiple outstanding low-confidence branches. On average, multipath reduces pipeline flushes by 40%, similarly to DMP. However, because multipath has very high overhead (200 wrong-path instructions per low-confidence branch, as shown in Figure 7), its average performance improvement is much less than that of DMP.

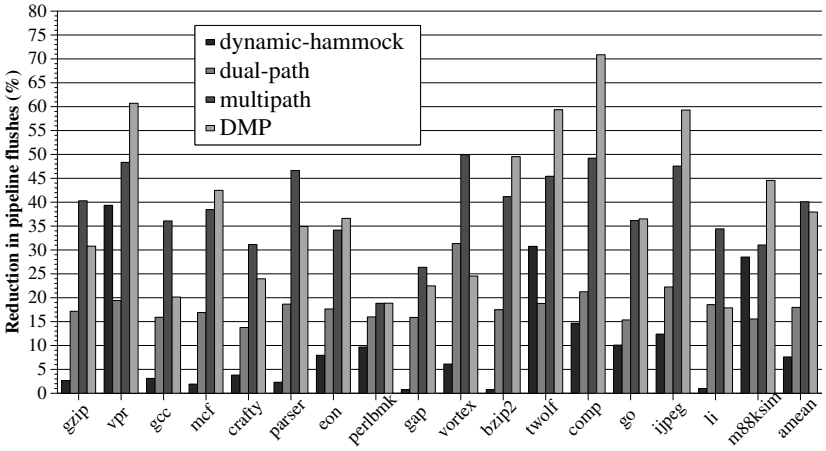

Figure 8. % reduction in pipeline flushes

5.2. Comparisons with Software Predication and Wish Branches

Figure 9 shows the execution time reduction over the baseline for limited software predication[12] and wish branches. Since the number of executed instructions is different in limited software predication and wish branches, we use the execution time metric for performance comparisons. Overall, limited software predication reduces execution time by 3.8%,wish branches by 6.4%, and DMP by 13.0%. In most benchmarks, wish branches perform better than predication because they can selectively enable predicated execution at run-time, thereby reducing the overhead of predication. Wish branches perform significantly better than limited software predication on vpr, parser, and ijpeg because they can be applied to loop branches.

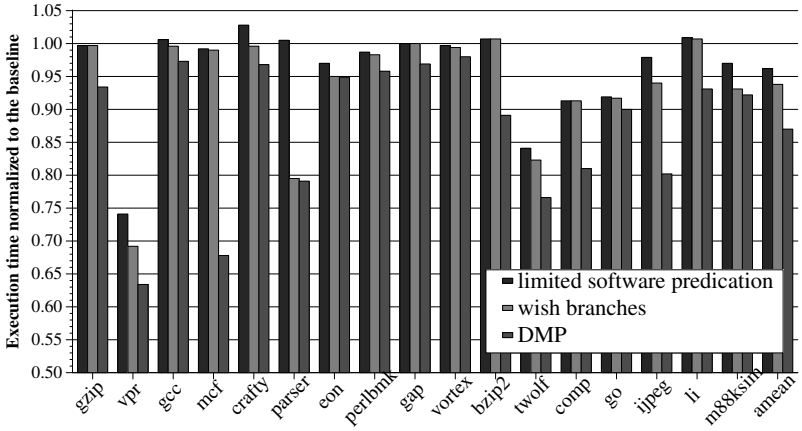

Figure 9. DMP vs. limited software predication and wish branches

There are some differences between previous results [21] and our results in the benefit of software predication and wish branches. The differences are due to the following: (1) our baseline processor already employs CMOVs which provide the performance benefit of predication for very small basic blocks, (2) ISA differences (Alpha vs. IA-64), (3) in our model of software predication, there is no benefit due to compiler optimizations that can be enabled with larger basic blocks in predicated code, (4) since wish branches dynamically reduce the overhead of software predication, they allow larger code blocks to be predicated, but we could not model this effect because Alpha ISA/compiler does not support predication.

Even though wish branches perform better than limited software predication, there is a large performance difference between wish branches and DMP. The main reason is that DMP can predicate frequently-hammocks, the majority of mispredicted branches in many benchmarks as shown in Figure 5. Only parser does not have many frequently-hammocks, so wish branches and DMP perform similarly for this benchmark.

[12] We call our software predication model "limited software predication" because we do not model compiler optimization effects enabled via if-conversion.

Figure 10 shows the performance improvement of DMP over the baseline if DMP is allowed to dynamically predicate: (1) only simple hammocks, (2) simple and nested hammocks, (3) simple, nested, frequently-hammocks, and (4) simple, nested, frequently-hammocks and loops. There is a large performance provided by the predication of frequently-hammocks as they are the single largest cause of branch mispredictions. Hence, DMP provides large performance improvements by enabling the predication of a wider range of CFGs than limited software predication and wish branches.

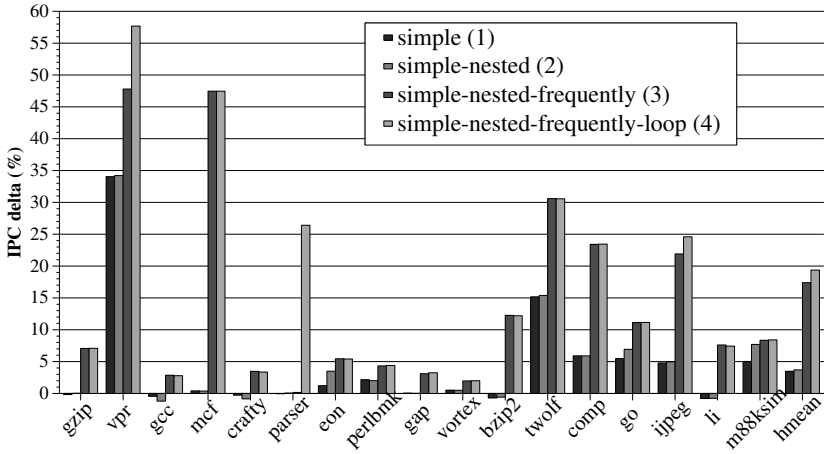

Figure 10. DMP performance when different CFG types are dynamically predicated

5.3. Analysis of the Performance Impact of Enhanced DMP Mechanisms

Figure 11 shows the performance improvement provided by the enhanced mechanisms in DMP. *Single-cfm* supports only a single CFM point for each diverge branch without any enhancements. Single-cfm by itself provides 11.4% IPC improvement over the baseline processor. *Multiple-cfm* supports more than one CFM point for each diverge branch as described in Section 3.3. Multiple-cfm increases the performance benefit of DMP for most benchmarks because it increases the probability of reaching a CFM point in dpred-mode and, hence, the likelihood of success of dynamic predication. *Mcfm-counter* supports multiple CFM points and also adopts the *Counter Policy* (Section 3.5). Counter Policy improves performance significantly in twolf, compress, and go; three benchmarks that have a high fraction of large frequently-hammock CFGs where the branch predictor sometimes deviates from the frequently executed paths. *Mcfm-counter-yield* also adopts the *Yield Policy* (Section 3.5) to exit dpred-mode early, increasing the performance benefit of DMP to 19.3%. Yield Policy is beneficial for vpr, mcf, twolf, compress, and go benchmarks. In these benchmarks, many diverge branches are control-flow dependent (i.e. nested) on other diverge branches, and control-flow dependent diverge branches are more likely to be mispredicted.

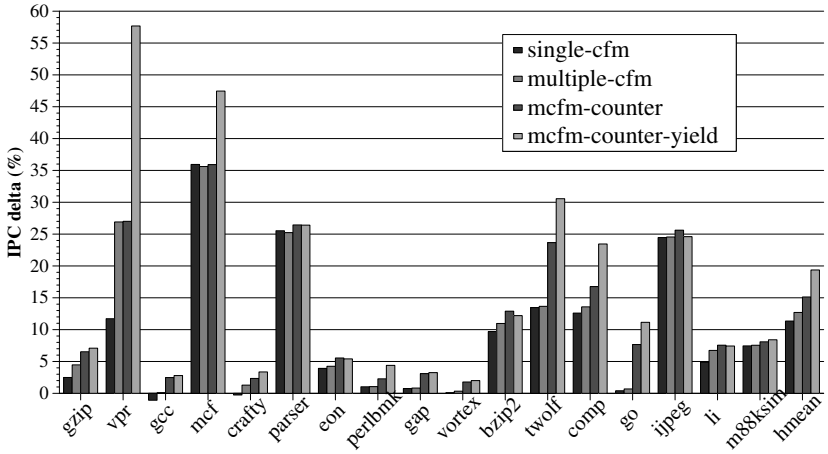

Figure 11. Performance impact of enhanced DMP mechanisms

5.4. Sensitivity to Microarchitecture Parameters

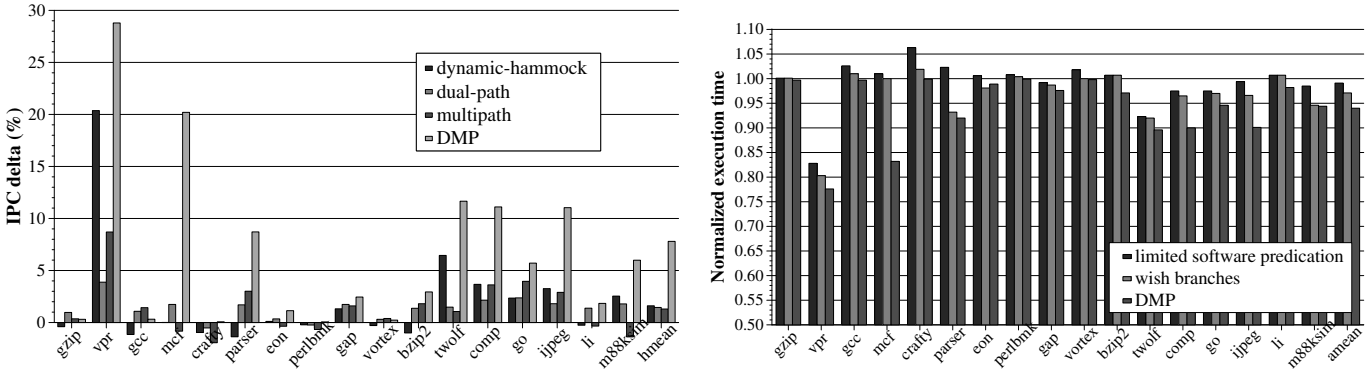

Figure 12. Performance comparison of DMP versus other paradigms on the less aggressive processor

5.4.1. Evaluation on the Less Aggressive Processor Figure 12 (left) shows the performance benefit for dynamic-hammock-predication, dual-path, multipath, and DMP on the less aggressive baseline processor and Figure 12 (right) shows the execution time reduction over the less aggressive baseline for limited software predication, wish branches, and DMP. Since the less aggressive processor incurs a smaller penalty for a branch misprediction, improved branch handling has less performance potential than in the baseline processor. However, DMP still provides 7.8% IPC improvement by reducing pipeline flushes by 30%, whereas dynamic-hammock-predication, dual-path and multipath improve IPC by 1.6%, 1.5%, and 1.3% respectively. Limited software predication reduces execution time by 1.0%, wish branches by 2.9%, and DMP by 5.7%.

5.4.2. Effect of a Different Branch Predictor We also evaluate DMP with a recently developed branch predictor, O-GEHL [36]. The O-GEHL predictor requires a complex hashing mechanism to index the branch predictor tables, but it effectively increases the global branch history length. As Figure 13 shows, replacing the baseline processor's perceptron predictor with a more complex, 64KB O-GEHL branch predictor (OGEHL-base) provides 13.8% performance improvement, which is smaller than the 19.3% performance improvement provided by implementing diverge-merge processing (perceptron-DMP). Furthermore, using DMP with an O-GEHL predictor (OGEHL-DMP) improves the average IPC by 13.3% over OGEHL-base and by 29% over our baseline processor. Hence, DMP still provides large performance benefits when the baseline processor's branch predictor is more complex and more accurate.

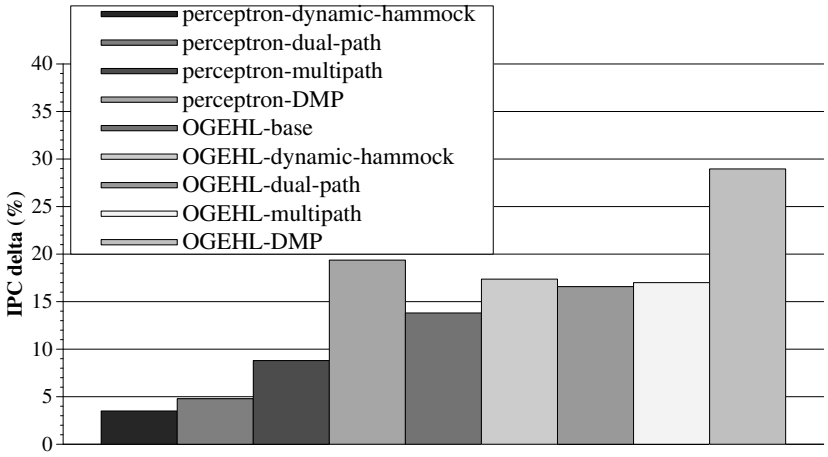

Figure 13. DMP performance with different branch predictors

5.4.3. Effect of Confidence Estimator Size Figure 14 shows the performance of dynamic-hammock-predication, dual-path, multipath and DMP with 512B, 2KB, 4KB, and 16KB confidence estimators and a perfect confidence estimator. Our baseline employs a 2KB enhanced

JRS confidence estimator [19], which has 14% PVN ($\simeq$ accuracy) and 70% SPEC ($\simeq$ coverage) [17].[13] Even with a 512-byte estimator, DMP still provides 18.4% performance improvement. The benefit of dual-path/multipath increases significantly with a perfect estimator because dual-path/multipath has very high overhead as shown in Figure 7, and a perfect confidence estimator eliminates the incurrence of this large overhead for correctly-predicted branches. However, even with a perfect estimator, dual-path/multipath has less potential than DMP because (1) dual-path is applicable to one low-confidence branch at a time (as explained previously in Section 5.1), (2) the overhead of dual-path/multipath is still much higher than that of DMP for a low-confidence branch because dual-path/multipath executes the same instructions twice/multiple times after a control-independent point in the program.

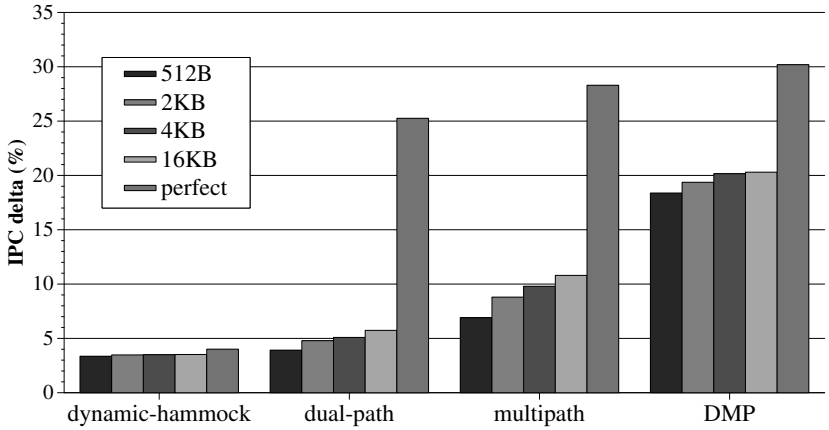

Figure 14. Effect of confidence estimator size on performance

5.5. Power Analysis

Figure 15 (left) shows the average increase/reduction due to DMP in the number of fetched/executed instructions, maximum power, energy, and energy-delay product compared to the baseline. Even though DMP has to fetch instructions from both paths of every dynamically predicated branch, the total number of fetched instructions decreases by 23% because DMP reduces pipeline flushes and thus eliminates the fetch of many wrong-path instructions. DMP executes 1% more instructions than the baseline due to the overhead of select-μops and predicated-FALSE instructions.

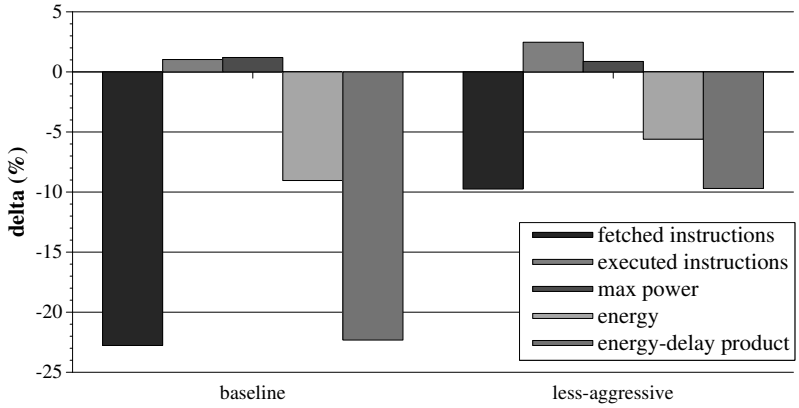

Figure 15. Power consumption comparison of DMP with the baseline processor (left) and less aggressive baseline processor (right)

Due to the extra hardware required to support DMP, maximum power consumption increases by 1.4%. However, because of the reduction in fetched instructions, energy consumption is reduced by 9.0%. Moreover, energy-delay product decreases by 22.3% because of both the performance improvement and energy reduction. Hence, although DMP increases hardware complexity, it actually increases energy-efficiency

[13] These numbers are actually lower than what was previously published [17] because our baseline branch predictor uses a different algorithm and has a much higher prediction accuracy than that of [17].

by reducing pipeline flushes due to branch mispredictions. DMP is an energy-efficient design even in the less aggressive processor configuration as Figure 15 (right) shows.

Table 6. Power and energy comparison of different branch processing paradigms

	Baseline processor						Less aggressive baseline processor					
	DMP	dyn-ham.	dual-path	multipath	SW-pred	wish br.	DMP	dyn-ham.	dual-path	multipath	SW-pred	wish br.
Max power Δ	1.4%	1.1%	1.2%	6.5%	0.1%	0.4%	0.9%	0.8%	0.8%	4.3%	0.1%	0.4%
Energy Δ	-9.0%	-0.7%	-2.2%	4.7%	-1.5%	-2.9%	-5.6%	-0.8%	1.1%	3.7%	-0.1%	-1.5%
Energy $\times$ Delay Δ	-22.3%	-0.9%	-7.0%	-4.3%	-1.8%	-6.1%	-9.7%	-0.5%	0.5%	2.2%	1.2%	-2.1%

Table 6 provides a power/energy comparison of the branch processing paradigms. DMP reduces energy consumption and energy-delay product much more than other approaches while it increases the maximum power requirements slightly more than the most relevant hardware techniques (dynamic-hammock-predication and dual-path). Note that multipath significantly increases both maximum power and energy consumption due to the extra hardware to support many outstanding paths.

6. Related Work

6.1. Related Work on Predication

Software predication has been studied intensively to reduce the branch misprediction penalty [27, 32, 43, 6] and to increase instruction-level parallelism [2]. However, in a real IA-64 implementation, predicated execution was found to provide a small (2%) performance improvement [9]. This small performance gain is due to the overhead and limitations of compile-time predication (described in Section 1), which sometimes offset the benefit of reducing the pipeline flushes due to branch mispredictions. Kim et al. [21] proposed wish branches to reduce the overhead of software predication by combining conditional branching and predication. DMP can predicate a larger set of CFGs than wish branches and it overcomes the major disadvantage of wish-branches: the requirement for a predicated ISA. Klauser et al. [22] proposed dynamic-hammock-predication for predicating only simple hammocks without support for predicated instructions in the ISA. DMP builds on dynamic-hammock-predication, but can predicate a much larger set of CFGs. Hence, as we showed in Section 5, DMP provides better performance and better energy efficiency.

Hyperblock formation [28] predicates frequently executed basic blocks based on profiling data, and it can predicate more complex CFGs than nested hammocks by tail duplication and loop peeling. The benefits of hyperblocks are that they increase the compiler's scope for code optimization and instruction scheduling (by enlarging basic blocks) in VLIW processors and they reduce branch mispredictions [27]. Unlike DMP, hyperblocks still require a predicated ISA, incur the overhead of software predication, are not adaptive to run-time changes in frequently executed control flow paths, and increase the code size [37].

6.2. Related Work on Dual-/Multi-path Execution

Heil and Smith [18] and Farrens et al. [15] proposed selective/limited dual path execution mechanisms. As we showed in Section 5, dual-path execution does not provide a performance improvement as significant as that of DMP because dual-path execution *always* wastes half of the fetch/execution resources even after a control-independent point in the program.

Selective eager execution (PolyPath) was proposed by Klauser et al. [24] as an implementation of multipath execution [33]. Multipath execution requires more hardware cost and complexity (e.g. multiple RATs/PCs/GHRs/RASs, logic to generate/manage path IDs/tags for multiple paths, logic to selectively flush the wrong paths, and more complex store-load forwarding logic that can support multiple outstanding paths) than DMP to keep multiple paths in the instruction window. As we have shown in Section 5.5, multipath execution significantly increases maximum power and energy consumption without providing as large performance improvements as that of DMP.

6.3. Related Work on Control Flow Independence

Several hardware mechanisms were proposed to exploit control flow independence [34, 35, 11, 8, 16]. These techniques aim to avoid flushing the processor pipeline if the processor is known to be at a control-independent point in the program when a mispredicted branch is resolved. In contrast to DMP, they require complex hardware to remove the control-dependent wrong-path instructions from the processor and to insert the control-dependent correct-path instructions into the pipeline after a branch misprediction. Hardware is also required to form correct data dependencies for the inserted correct path instructions. Furthermore, control-independent instructions that are data-dependent on the inserted or removed instructions have to be re-scheduled and re-executed with the correct data dependencies and after the processor finishes fetching and renaming the new inserted instructions. The logic required for ensuring correct data dependencies for both control-dependent and control-independent instructions is complicated as Rotenberg et al. pointed out [34].

Collins et al. [12] introduced dynamic reconvergence prediction, a hardware-based technique to identify control reconvergence points (i.e. our CFM points) without compiler support. This technique can be combined with DMP (so that CFM points are discovered at run-time rather than compile-time) and any of the mechanisms that exploit control-flow independence.

7. Conclusion and Future Work

This paper proposed the diverge-merge processor (DMP) as an efficient architecture for compiler-assisted dynamic predicated execution. DMP dynamically predicates hard-to-predict instances of statically-selected diverge branches. The major contributions of the diverge-merge processing concept are:

1. DMP enables the dynamic predication of branches that result in *complex control-flow graphs* rather than limiting dynamic predication to simple hammock branches. The key insight is that most control-flow graphs look and behave like simple hammock (if-else) structures when only frequently executed paths in the graphs are considered. Therefore, DMP can eliminate branch mispredictions due to a much larger set of branches than previous predication techniques such as software predication and dynamic hammock predication.
2. DMP concurrently overcomes the three major limitations of software predication (described in Section 1).
3. DMP eliminates branch misprediction flushes much more efficiently (i.e. with less instruction execution overhead) than alternative approaches, especially dual-path and multipath execution (as shown in Table 1 and Figure 7).

Our results show that DMP outperforms an aggressive baseline processor with a very large branch predictor by 19.3% while consuming 9.0% less energy. Furthermore, DMP provides higher performance and better energy-efficiency than dynamic hammock predication, dual-path/multipath execution, software predication, and wish branches.

The proposed DMP mechanism still requires some ISA support. A cost-efficient hardware mechanism to detect diverge branches and CFM points at run-time would eliminate the need to change the ISA. Developing such mechanisms is part of our future work. The results presented in this paper are based on our initial implementation of DMP using relatively simple compiler and hardware heuristics/algorithms. The performance improvement provided by DMP can be increased further 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. On the hardware side, better confidence estimators are worthy of research since they critically affect the performance benefit of dynamic predication.

Acknowledgments

Special thanks to Chang Joo Lee for the support he provided in power modeling. We thank Paul Racunas, Veynu Narasiman, Nhon Quach, Derek Chiou, Eric Sprangle, Jared Stark, and members of the HPS group for comments and suggestions. 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] P. S. Ahuja, K. Skadron, M. Martonosi, and D. W. Clark. Multipath execution: opportunities and limits. In *ICS-12*, 1998.

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

[3] D. I. August, D. A. Connors, J. C. Gyllenhaal, and W. W. Hwu. Architectural support for compiler-synthesized dynamic branch prediction strategies: Rationale and initial results. In *HPCA-3*, 1997.

[4] D. I. August, W. W. Hwu, and S. A. Mahlke. A framework for balancing control flow and predication. In *MICRO-30*, 1997.

[5] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *ISCA-27*, 2000.

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

[7] S. Chaudhry, P. Caprioli, S. Yip, and M. Tremblay. High-performance throughput computing. *IEEE Micro*, 25(3):32–45, May 2005.

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

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

[10] Y. Chou, B. Fahs, and S. Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. In *ISCA-31*, 2004.

[11] Y. Chou, J. Fung, and J. P. Shen. Reducing branch misprediction penalties via dynamic control independence detection. In *ICS-13*, 1999.

[12] J. D. Collins, D. M. Tullsen, and H. Wang. Control flow optimization via dynamic reconvergence prediction. In *MICRO-37*, 2004.

[13] A. Cristal, O. J. Santana, F. Cazorla, M. Galluzzi, T. Ramirez, M. Pericas, and M. Valero. Kilo-instruction processors: Overcoming the memory wall. *IEEE Micro*, 25(3):48, May 2005.

[14] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. Technical Report RC14756, IBM, revised March 1991.

[15] M. Farrens, T. Heil, J. E. Smith, and G. Tyson. Restricted dual path execution. Technical Report CSE-97-18, University of California at Davis, Nov. 1997.

[16] A. Gandhi, H. Akkary, and S. Srinivasan. Reducing branch misprediction penalty via selective recovery. In *HPCA-10*, 2004.

[17] D. Grunwald, A. Klauser, S. Manne, and A. Pleszkun. Confidence estimation for speculation control. In *ISCA-25*, 1998.

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

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

[20] D. A. Jiménez and C. Lin. Dynamic branch prediction with perceptrons. In *HPCA-7*, 2001.

[21] H. Kim, O. Mutlu, J. Stark, and Y. N. Patt. Wish branches: Combining conditional branching and predication for adaptive predicated execution. In *MICRO-38*, 2005.

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

[23] A. Klauser and D. Grunwald. Instruction fetch mechanisms for multipath execution processors. In *MICRO-32*, 1999.

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

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

[26] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In *ISCA-19*, 1992.

[27] S. A. Mahlke, R. E. Hank, R. A. Bringmann, J. C. Gyllenhaal, D. M. Gallagher, and W. W. Hwu. Characterizing the impact of predicated execution on branch prediction. In *MICRO-27*, 1994.

[28] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *MICRO-25*, 1992.

[29] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *HPCA-9*, 2003.

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

[31] J. C. H. Park and M. Schlansker. On predicated execution. Technical Report HPL-91-58, Hewlett-Packard Labs, Palo Alto CA, May 1991.

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

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

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

[35] E. Rotenberg and J. Smith. Control independence in trace processors. In *MICRO-32*, 1999.

[36] A. Seznec. Analysis of the O-GEometric History Length branch predictor. In *ISCA-32*, 2005.

[37] J. W. Sias, S. Ueng, G. A. Kent, I. M. Steiner, E. M. Nystrom, and W. W. Hwu. Field-testing IMPACT EPIC research results in Itanium 2. In *ISCA-31*, 2004.

[38] B. Simon, B. Calder, and J. Ferrante. Incorporating predicate information into branch predictors. In *HPCA-9*, 2003.

[39] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark. Branch prediction, instruction-window size, and cache size: Performance trade-offs and simulation techniques. *ACM Transactions on Computer Systems*, 48(11):1260–1281, Nov. 1999.

[40] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. In *ISCA-29*, 2002.

[41] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual flow pipelines. In *ASPLOS-XI*, 2004.

[42] J. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Technical White Paper*, Oct. 2001.

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

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

[45] N. J. Warter, S. A. Mahlke, W. W. Hwu, and B. R. Rau. Reverse if-conversion. In *PLDI*, 1993.