Profile-assisted Compiler Support for Dynamic Predication in Diverge-Merge Processors

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

Dynamic predication has been proposed to reduce the branch misprediction penalty due to hard-to-predict branch instructions. A recently proposed dynamic predication architecture, the diverge-merge processor (DMP), provides large performance improvements by dynamically predicating a large set of complex control-flow graphs that result in branch mispredictions. DMP requires significant support from a profiling compiler to determine which branch instructions and control-flow structures can be dynamically predicated. However, previous work on dynamic predication did not extensively examine the tradeoffs involved in profiling and code generation for dynamic predication architectures.

This paper describes compiler support for obtaining high performance in the diverge-merge processor. We describe new profile-driven algorithms and heuristics to select branch instructions that are suitable and profitable for dynamic predication. We also develop a new profile-based analytical cost-benefit model to estimate, at compile-time, the performance benefits of the dynamic predication of different types of control-flow structures including complex hammocks and loops. Our evaluations show that DMP can provide 20.4% average performance improvement over a conventional processor on SPEC integer benchmarks with our optimized compiler algorithms, whereas the average performance improvement of the best-performing alternative simple compiler algorithm is 4.5%. We also find that, with the proposed algorithms, DMP performance is not significantly affected by the differences in profile- and run-time input data sets.

1. Introduction

Branch misprediction penalty is an important limitation for high-performance processors, even after significant research in branch predictors. Predication eliminates branches and therefore avoids the misprediction penalty, but it requires significant modifications to the ISA and it can degrade performance when a statically if-converted branch could have been correctly predicted. Instances of the same static branch could be easy or hard to predict during different phases of a program execution. Dynamic predication allows the processor to predicate instructions without requiring a predicated ISA and to choose when to predicate at run-time [15]. A processor that supports dynamic predication executes both paths of a branch until it reaches the control-flow convergence point of the branch. Instructions on both paths are executed but only the correct-path instructions update the architectural state.

Dynamic hammock predication was proposed for dynamic predication of simple hammock structures (simple `if` and `if-else` structures with no intervening control flow instructions) [15]. The Diverge-Merge Processor (DMP) architecture extends the dynamic predication concept to more complex code structures [12]. The key improvement of DMP over previous work is its ability to predicate control-flow shapes that are not simple hammocks statically but that look like simple hammocks when only frequently-executed control flow paths at run-time are considered. These control-flow shapes are termed as *frequently-hammock*s.

In the DMP architecture, branches that can be dynamically predicated (i.e. *diverge branches*) and the corresponding control-flow convergence/merge points (*CFM-points*) are identified by the compiler and conveyed to the hardware through the ISA. Diverge branches can be parts of either simple hammocks or frequently-hammocks. How the compiler selects diverge branches and CFM points and how the processor chooses when to predicate them at run-time are critical factors that determine the performance of dynamic predication in a DMP processor. Previous work did not explore compiler algorithms/heuristics and profiling mechanisms used to select diverge branches and CFM points. In this paper, we describe the compiler and profiling algorithms for a DMP processor and explore the tradeoffs involved in the design of these algorithms. We evaluate the impact of these algorithms on the performance of a DMP processor and provide insights into what is important to consider in the design of such algorithms.

This paper makes the following contributions:

1. To our knowledge, this is the first paper that develops detailed code generation algorithms for dynamic predication architectures. We provide insights into the design of a profiler/compiler targeted for a DMP architecture. We explain and quantitatively analyze the tradeoffs involved in making the design choices for the profiler/compiler, a key component of the DMP architecture.
2. We propose an analytical, profile-driven cost-benefit model for dynamic predication used by the compiler to decide candidate branches for dynamic predication. The proposed model can also be used for understanding the behavior of DMP and improving its microarchitecture.
3. We analyze the sensitivity of a DMP architecture to differences in profile-time and run-time input sets. We explain and analyze the issues involved in profiling for the DMP architecture.

2. Background

2.1. Dynamic Predication: Simple Hammocks [15]

Figure 1 shows the control-flow graph (CFG) of a simple hammock branch and the dynamically predicated instructions. Hammock branches are identified at run-time or marked by the compiler. When the processor fetches a hammock branch, it estimates whether or not the branch is hard to predict using a branch confidence estimator [9]. If the branch has low confidence, the processor dynamically predicates instructions on both paths of the branch (i.e. the processor

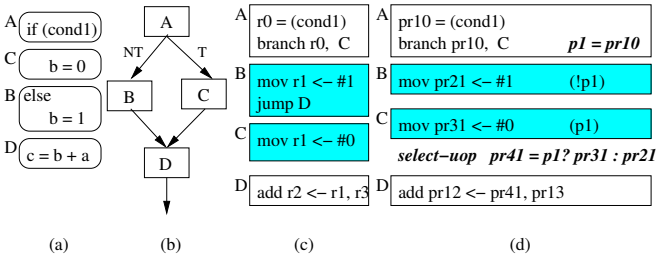

Figure 1. Simple hammock example: (a) source code (b) CFG (c) assembly code (d) predicated instructions after register renaming

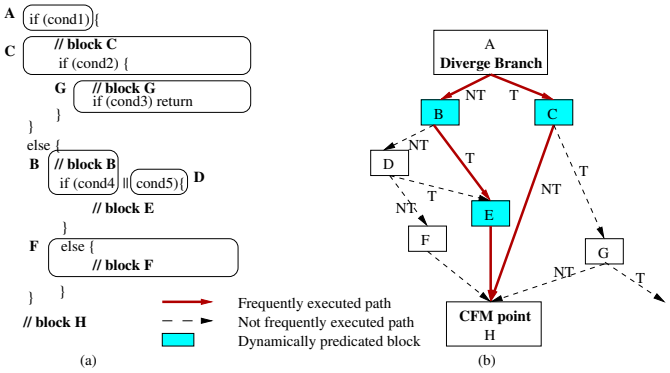

Figure 2. Complex CFG example: (a) source code (b) CFG

enters *dynamic predication mode (dpred-mode)* in Kim et al.'s terminology [12]). The processor generates a predicate id using the branch condition, and instructions inside the hammock are assigned the generated predicate id. When the hammock branch is resolved, the predicate id is also resolved. Instructions on the wrong path (i.e. predicated-FALSE instructions) become NOPs after the branch is resolved, and they do not update the architectural state. When the processor reaches a control reconvergence point after fetching both paths of the branch, the processor inserts c-moves [11] or select-μops [24] to reconcile the register data values produced on either side of the hammock. Select-μops are similar to the ϕ-functions in the static single-assignment (SSA) form [5].

2.2. Dynamic Predication: DMP [12]

DMP extends dynamic predication to complex CFGs. Figure 2 shows a CFG example to illustrate the key mechanism of DMP. The processor considers frequently executed paths at run-time, so it can dynamically predicate blocks B, C, and E. To simplify the hardware, DMP uses some control-flow information provided by the compiler. 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 CFG, a point 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 CFG* of the program but they are not necessarily simple hammock branches that *require* the CFG to be hammock-shaped. The compiler also identifies at least one CFM point associated with the 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 CFM point.

The DMP microarchitecture fetches both paths after a low-confidence diverge branch and dynamically predicates instructions between the diverge branch and one of its CFM points during dpred-mode. On each path, the processor follows the branch predictor outcomes until it reaches a CFM point. After the processor reaches the same CFM point on both paths, it exits dpred-mode and starts to fetch from only one path. When DMP exits dpred-mode, select-μops are inserted to reconcile the register data values that are produced on either side of the "dynamic hammock."

DMP can also dynamically predicate loop branches. The benefit of predicating loop branches is that 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. Further explanations about the diverge loop behavior are provided in Section 5.1 when we discuss compiler heuristics for choosing diverge loops.

3. Compiler Algorithms for DMP Architectures

The compiler marks the diverge branches and their respective CFM points in a DMP binary. At run-time, the processor decides whether or not to enter dpred-mode based on the confidence estimation for a diverge branch. The hardware has relatively more accurate dynamic information on whether or not a diverge branch is likely to be mispredicted. However, it is difficult for the hardware to determine (1) the CFM point of a branch, (2) whether or not dynamically predicating a diverge branch would provide performance benefit.[1] The performance benefit of dynamic predication is strongly dependent on the number of instructions between a diverge branch and its corresponding CFM points (similarly to static predication [19, 17, 23, 18]). In frequently-hammocks, the probability that both paths after a diverge branch reach a CFM point could be another factor that determines whether or not dynamically predicating the diverge branch would be beneficial for performance. Since the compiler has easy access to both CFG information and profiling data to estimate frequently executed paths, it can estimate which branches and CFM points would be good candidates to be dynamically predicated. Thus, in this section, we develop profile-driven compiler algorithms to solve the following new problems introduced by DMP processors:

1. DMP introduces a new CFG concept: frequently-hammocks. We develop a compiler algorithm to find frequently-hammocks and their corresponding CFM points.
2. DMP requires the selection of diverge branches and corresponding CFM points that would improve performance when dynamically predicated. We develop compiler algorithms to determine which branches should be selected as diverge branches and which CFM point(s) should be selected as corresponding CFM point(s). Simple algorithms and heuristics are developed in this section and a more detailed cost-benefit model is presented in Section 4.

3.1. Diverge Branch Candidates

We consider four types of diverge branches based on the CFG types they belong to. Simple hammock (Figure 3a) is an `if` or `if-else` structure that does not have any control-flow instructions

[1]The hardware could measure the usefulness of dynamic predication for each branch at run-time, but the previously proposed DMP implementation [12] does not support such a feedback scheme due to the hardware cost associated with such a scheme.

inside the hammock. Nested hammock (Figure 3b) is an `if-else` structure that has multiple levels of nested branches. Frequently-hammock (Figure 3c) is a complex CFG, which is not a simple/nested hammock, but becomes a simple hammock if we consider only frequently executed paths. Loop (Figure 3d) is a cyclic CFG (`for`, `do-while`, or `while` structure) with the diverge branch as a loop exit branch.

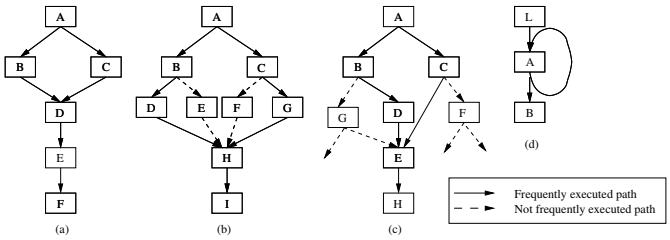

Figure 3. Types of CFGs: (a) simple hammock (b) nested hammock (c) frequently-hammock (d) loop. The branch at the end of block A is a possible diverge branch.

We also classify CFM points into two categories: exact and approximate. *Exact CFM points* are those that are always reached from the corresponding diverge branch, independently of the actually executed control-flow paths between the branch and the CFM point. In other words, an exact CFM point is the immediate post-dominator (IPOSDOM) of the diverge branch. *Approximate CFM points* are those that are reached from the corresponding diverge branch only on the frequently-executed paths. Simple and nested hammocks and single-exit loops have only exact CFM points. Frequently-hammocks have approximate CFM points.

3.2. Algorithm to Select Simple/Nested Hammock Diverge Branches and Exact CFM Points

Algorithm 1 (Alg-exact) describes how to find and select simple and nested hammock diverge branches that have exact CFM points. Simple and nested hammocks have strictly one exact CFM point, which is the IPOSDOM of the branch. We use Cooper et al.'s algorithm [4] to find the IPOSDOM. Our algorithm uses the number of instructions and the number of conditional branches between the branch and the CFM point to select diverge branches among the possible candidates.

Algorithm 1 Finding and selecting simple/nested-hammock diverge branches and exact CFM points (Alg-exact)

for each conditional branch B **do**
  Compute $IPOSDOM(B)$ of B
  $num\_instr \leftarrow$ maximum number of static instructions on any path from B to $IPOSDOM(B)$
  $num\_cbr \leftarrow$ maximum number of conditional branches on any path from B to $IPOSDOM(B)$
  if ($num\_instr \leq MAX\_INSTR$) and ($num\_cbr \leq MAX\_CBR$) **then**
    mark B as a diverge branch candidate with $CFM = IPOSDOM(B)$
  end if
end for

This algorithm eliminates candidates that can reconverge only after a large number of instructions (MAX_INSTR) on any path. This is because the benefit of DMP processors comes from fetching and possibly executing instructions following the CFM point after dynamically predicating both paths of a diverge branch. Such control-independent instructions do not have to be flushed when the diverge branch is resolved. If either the taken or the not-taken path of the diverge branch is too long, the processor's instruction window is likely to be filled before reaching the CFM point, thereby reducing the potential benefit of DMP. Additionally, instructions on the wrong path of the dynamically-predicated branch consume machine resources, increasing the overhead of predication. Therefore, a branch with a potentially long wrong path before the CFM point (i.e. a branch that has a large number of instructions between itself and its CFM point) is not a good candidate for dynamic predication and is not selected as a diverge branch by our algorithm.

Alg-exact also eliminates candidates with a large number of conditional branches (MAX_CBR) on any path from the branch to the CFM point. DMP can enter dpred-mode for only one branch at a time. Limiting the number of conditional branches that are allowed between a diverge branch and its CFM point reduces the likelihood of another low-confidence branch occurring on a predicated path. Since the number of conditional branches is correlated with the number of instructions, we conservatively use $MAX\_CBR = MAX\_INSTR/10$ in all experiments. We experiment with different values for MAX_INSTR.

3.3. Algorithm to Select Frequently-hammock Diverge Branches and Approximate CFM Points

Algorithm 2 (Alg-freq) describes our algorithm for finding and selecting frequently-hammock diverge branches and their approximate CFM points. The algorithm uses edge profiling information to determine frequently executed paths.

While traversing the CFG to compute paths after a branch, only directions (taken/not-taken) that were executed with at least MIN_EXEC_PROB during the profiling run are followed. This threshold (set to 0.001) eliminates the exploration of extremely infrequently executed paths during the search for paths that merge at CFMs, reducing the processing time of the algorithm.

In addition to MAX_INSTR and MAX_CBR, the algorithm for selecting frequently-hammocks uses the probability of merging at each CFM point (MIN_MERGE_PROB) and the number of CFM points (MAX_CFM). The CFM point candidates with the highest probability of being reached on both paths during the profiling run are selected by our algorithm because dynamic predication provides more benefit if both paths of a diverge branch reach a corresponding CFM point.[2] If the profiled probability of reaching a CFM point candidate is lower than a threshold (MIN_MERGE_PROB), the CFM point candidate is not selected as a CFM point. Selecting multiple CFM points for a diverge branch increases the likelihood that the predicated paths after a diverge branch will actually reconverge and thus increases the likelihood that dynamic predication would provide performance benefits. Since we found that using three CFM

[2]If both paths after the dynamically-predicated diverge branch do not merge at a CFM point, DMP could still provide performance benefit. In that case, the benefit would be similar to that of dual-path execution [8].

Algorithm 2 Finding and selecting frequently-hammock diverge branches and approximate CFM points (Alg-freq)

```
1:  for each conditional branch B executed during profiling do
2:      Compute IPOSDOM(B) of B
3:      With a working list algorithm, compute all paths starting
        from B, up to reaching IPOSDOM(B) or MAX_INSTR
        instructions or MAX_CBR conditional branches, following
        only branch directions with profiled frequency ≥
        MIN_EXEC_PROB.
4:      for each basic block X reached on both the taken and the not-
        taken directions of B do
5:          p_T(X) ← edge-profile-based probability of X being
            reached on the taken direction of B
6:          p_NT(X) ← edge-profile-based probability of X being
            reached on the not-taken direction of B
7:          probability of merging at X ← p_T(X) * p_NT(X)
8:          if (probability of merging at X ≥
                MIN_MERGE_PROB) then
9:              add X as a CFM point candidate for B
10:         end if
11:     end for
12:     select up to MAX_CFM CFM point candidates for B, the
        ones with the highest probability of merging at X
13: end for
```

points is enough to get the full benefit of our algorithms, we set $MAX\_CFM = 3$.

3.3.1. A chain of CFM Points Figure 4 shows a possible CFG with two CFM point candidates, C and D, for the branch at A. The DMP processor stops fetching from one path when it reaches the first CFM point in dpred-mode. Since the taken path of the diverge branch candidate at A always reaches C before it reaches D, even if both C and D are selected as CFM points, dynamic predication would always stop at C. D would never be reached by both dynamically-predicated paths of the branch at A in dpred-mode, and thus choosing D as a CFM point does not provide any benefit if C is chosen as a CFM point. Therefore, the compiler should choose either C or D as a CFM point, but not both. In general, if a CFM point candidate is on any path to another CFM point candidate, we call these candidates *a chain of CFM points*. The compiler identifies chains of CFM point candidates based on the list of paths from the diverge branch to each CFM point candidate, generated by Alg-freq. Then, the compiler conservatively chooses only one CFM point in the chain, the one with the highest probability of merging.[3]

3.4. 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. Our experiments found that always predicating hammocks that execute fewer than 10 instructions on each path, that have a probability of merging of at least 95%, and that have a branch misprediction rate of at least 5% provides the best performance.

Note that, with this heuristic, diverge branch-CFM point pairs that are identified as *short hammocks* are always predicated, unlike regular hammocks. Therefore, any other CFM point candidates found for the same diverge branch that do not qualify as short hammocks are not selected as CFM points.

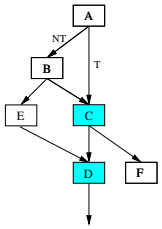

Figure 4. Example of a chain of CFM points

3.5. 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.

4. Compile-Time Cost-Benefit Analysis of Dynamic Predication

In the basic algorithms presented in Section 3 (Alg-exact and Alg-freq), the compiler uses several simple heuristics to select diverge branches and CFM points that are likely to provide performance benefit during dynamic predication. These algorithms require the MAX_INSTR, MAX_CBR, and MIN_MERGE_PROB thresholds to be optimized. Determining an effective combination of these parameters may require several iterations. In this section, we present an analytical cost-benefit model to select diverge branches and CFM points whose dynamic predication is likely to be beneficial for overall performance. The cost-benefit model still uses Alg-exact and Alg-freq to find diverge branch and CFM point candidates, but instead of filtering candidates with the compile-time fixed MIN_MERGE_PROB, MAX_INSTR, and MAX_CBR parameters, it performs a profile-driven cost-benefit analysis.[4]

[3] When there is a chain of CFM points, the $probability\ of\ merging\ at\ X$ in Alg-freq has to be modified to compute the probability of both paths of the diverge branch *actually merging* at X for the first time, instead of just *reaching* X. For the diverge branch candidate A in Figure 4, $probability\ of\ merging\ at\ C = p_T(C) * p_{NT}(C) = 1 * P(BC) = P(BC)$, where P(BC) is the edge probability from B to C. In contrast, $probability\ of\ merging\ at\ D = p_T(D) * p_{NT}(D) = P(CD) * P(BE)$ because if the not-taken path of the branch at A takes BC, the actual merging point would be C instead of D.

[4] In order to use Alg-exact and Alg-freq, the compiler still needs values for MAX_INSTR and MAX_CBR because these parameters also decide the compiler scope for the CFG analysis. In our cost-benefit model, we use $MAX\_INSTR = 200$ and $MAX\_CBR = 20$, which we found to be large enough to enable the analysis of all CFGs that can profit from dynamic predication.

4.1. Simple/Nested Hammocks

During dpred-mode, DMP always incurs some performance overhead in terms of execution cycles. The overhead of dynamic predication ($dpred\_overhead$) is due to the fetch and possible execution of useless (i.e. wrong-path) instructions. We describe how a profiling compiler can model the overhead of dynamic predication and make decisions as to whether or not dynamically predicating a branch instruction would be beneficial for performance.

There are two cases for which the cost of dynamic predication of a branch is different. First, if a diverge branch would actually have been correctly predicted, entering dpred-mode for that branch results only in overhead ($dpred\_overhead$) without providing any benefit. Second, if a diverge branch would actually have been mispredicted, entering dpred-mode for that branch results in both overhead ($dpred\_overhead$) and performance benefit that is equivalent to saving the branch misprediction penalty ($misp\_penalty$ cycles). Hence, the overall cost of dynamic predication ($dpred\_cost$) in terms of cycles can be computed as:

$$dpred\_cost = dpred\_overhead * P(enter\_dpred\_corr\_pred) + (dpred\_overhead - misp\_penalty) * P(enter\_dpred\_misp) \quad (1)$$

$$P(enter\_dpred\_corr\_pred) = 1 - Acc\_Conf \quad (2)$$

$$P(enter\_dpred\_misp) = Acc\_Conf \quad (3)$$

$dpred\_overhead$: Overhead of dynamic predication in cycles
$P(enter\_dpred\_corr\_pred)$: Probability of entering dpred-mode when a branch is correctly predicted
$P(enter\_dpred\_misp)$: Probability of entering dpred-mode when a branch is mispredicted
$misp\_penalty$: Machine-specific branch misprediction penalty in cycles
Acc_Conf: The accuracy of the confidence estimator (i.e. the fraction of low-confidence branches that are actually mispredicted)

The compiler decides to select a branch as a diverge branch if the cost of dynamic predication, as determined using Equation (1), is less than zero (i.e. if the benefit of dynamic predication is positive in terms of execution cycles):

$$\text{Select a branch as a diverge branch if } dpred\_cost < 0 \quad (4)$$

Note that the probability of entering dpred-mode when a branch is correctly predicted versus when it is mispredicted is a function of the accuracy of the hardware confidence estimator [9]. Confidence estimator accuracy (defined as the percentage of low-confidence branches that are actually mispredicted, i.e. PVN [6]) is usually between 15%-50% and is dependent on confidence estimator parameters such as the threshold values used in the design [6]. In the calculation of the cost of dynamic predication, the compiler can use the average accuracy of the confidence estimator based on the set of profiled benchmarks or it can obtain the accuracy of the confidence estimator for each individual application and use that per-application accuracy. In our analysis the compiler uses one accuracy value ($Acc\_Conf = 40\%$) for all applications.[5]

4.1.1. Estimation of the overhead of dynamic predication

To calculate the overhead of dynamic predication ($dpred\_overhead$), the compiler first estimates the number of instructions fetched between a diverge branch candidate and the corresponding CFM point ($N(dpred\_insts)$). The compiler can estimate $N(dpred\_insts)$ in three different ways: (1) based on the most frequently-executed two paths (using profile data), (2) based on the longest path between the diverge branch candidate and the CFM point, (3) based on the average number of instructions obtained using edge profile data. Equations 5-11 show how the compiler calculates $N(dpred\_insts)$ with these three different methods using the example presented in Figure 2. Note that the most frequently executed paths are shaded in Figure 2. In the equations, N(X) is the number of instructions in block X, and P(XY) is the edge probability from basic block X to Y.[6] In our experiments, we evaluate methods 2 and 3.

$$N(dpred\_insts) = N(BH) + N(CH) \quad (5)$$

$N(BH)$: Estimated number of insts from block B to the beginning of block H
$N(CH)$: Estimated number of insts from block C to the beginning of block H

(Method 1) Based on the most frequently-executed two paths:

$$N(BH) = N(B) + N(E) \quad (6)$$

$$N(CH) = N(C) \quad (7)$$

(Method 2) Based on the longest possible path:

$$N(BH) = MAX\{N(B) + N(D) + N(F), N(B) + N(D) + N(E), N(B) + N(E)\} \quad (8)$$

$$N(CH) = N(C) + N(G) \quad (9)$$

(Method 3) Based on the edge profile data (i.e. average number of instructions)

$$N(BH) = N(B) + P(BE) * N(E) + P(BD) * P(DE) * N(E) + P(BD) * N(D) + P(BD) * P(DF) * N(F) \quad (10)$$

$$N(CH) = N(C) + P(CG) * N(G) \quad (11)$$

Because not all of the instructions fetched in dpred-mode are useless, the compiler also estimates the number of instructions that are actually useful (i.e. those that are on the correct path). The number of instructions on the correct path in dpred-mode ($N(useful\_dpred\_insts)$) is calculated as follows. $N(BH)$ and $N(CH)$ can be calculated with any of above three methods.

$$N(useful\_dpred\_insts) = P(AB) * N(BH) + P(AC) * N(CH) \quad (12)$$

Once the compiler has computed $N(dpred\_insts)$ and $N(useful\_dpred\_insts)$, it can calculate $dpred\_overhead$. We calculate $dpred\_overhead$ in terms of fetch cycles. The actual cost of dynamic predication is the sum of its fetch overhead and execution overhead. Unfortunately, modeling the execution overhead is very complicated in an out-of-order processor due to the dataflow-based dynamic execution (which requires an analytical model of benchmark-dependent data dependence behavior as well as a model of dynamic events that affect execution). Furthermore, DMP does not execute predicated-FALSE instructions after the predicate value is known, so the execution overhead is likely not as high as the fetch overhead. Therefore, we model only the fetch overhead of dynamic

[5] Note that there is a trade-off between coverage (of mispredicted branches) and accuracy in confidence estimators. We found that the cost-benefit model is not sensitive to reasonable variations in Acc_Conf values (20%-50%). We do not present the results of varying Acc_Conf due to space limitations.

[6] Edge profiling assumes that the direction taken by a branch is independent of the direction taken by a previous branch, which is not always accurate. However, we use edge profiling due to its simplicity and short run-time.

predication in our cost-benefit analysis. The overhead of dynamically predicating a branch in terms of fetch cycles is thus calculated as:

$$N(useless\_dpred\_insts) = N(dpred\_insts) - N(useful\_dpred\_insts) \quad (13)$$

$$dpred\_overhead = N(useless\_dpred\_insts)/fw \quad (14)$$

fw: Machine-specific instruction fetch width
$useless\_dpred\_insts$: Useless instructions fetched during dpred-mode

Combining Equation (14) with Equations (1) and (4) gives us the final equation used by the compiler to decide whether or not a branch should be selected as a diverge branch:

Select a branch as a diverge branch if

$$\{N(useless\_dpred\_insts)/fw\} * P(enter\_dpred\_corr\_pred) + \{(N(useless\_dpred\_insts)/fw) - misp\_penalty\} * P(enter\_dpred\_misp) < 0 \quad (15)$$

4.2. Frequently-hammocks

The overhead of predicating frequently-hammocks is usually higher than that of predicating simple or nested hammocks. With a frequently-hammock, the processor might not reach the corresponding CFM point during dpred-mode. In that case, the processor wastes half of the fetch bandwidth to fetch useless instructions until the diverge branch is resolved. On the other hand, if the processor reaches the CFM point in dpred-mode, the predication overhead of frequently-hammocks is the same as that of simple/nested hammocks, as calculated in Equation (14). Therefore, we use the following equation to calculate the dynamic predication overhead of a frequently-hammock:

$$dpred\_overhead = \{1 - P(merge)\} * \{branch\_resol\_cycles/2\} + P(merge) * \{N(useless\_dpred\_insts)/fw\} \quad (16)$$

$P(merge)$: The probability of both paths after the candidate branch merging at the CFM point (based on edge profile data)
$branch\_resol\_cycles$: The time (in cycles) between when a branch is fetched and when it is resolved (i.e. $misp\_penalty$)

The resulting $dpred\_overhead$ is plugged into Equations (1) and (4) to determine whether or not selecting a frequently-hammock branch as a diverge branch would be beneficial for performance.

4.3. Diverge Branches with Multiple CFM Points

So far, we have discussed how the compiler selects diverge branches assuming that there is only one CFM point for each diverge branch. However, in frequently-hammocks, there are usually multiple CFM point candidates for a branch. After reducing the list of CFM point candidates according to Section 3.3.1, the overhead of dynamically predicating a diverge branch with multiple CFM points is computed assuming all CFM points (X_i) are independent:

$$dpred\_overhead = \{\sum_i N(useless\_dpred\_insts(X_i)) * P(merge\ at\ X_i)\}/fw + \{1 - \sum_i P(merge\ at\ X_i)\} * \{branch\_resolution\_cycles/2\} \quad (17)$$

$N(useless\_dpred\_insts(x))$: $useless\_dpred\_insts$ assuming x is the only CFM point of the diverge branch candidate

If the diverge branch candidate satisfies Equations (1) and (4) after using the $dpred\_overhead$ developed in Equation (17), the branch is selected as a diverge branch with its reduced list of CFM points.

4.4. Limitations of the Model

Note that we make the following assumptions to simplify the construction of the cost-benefit analysis model:

1. The processor can fetch fw ($fetchwidth$) number of instructions all the time. There are no I-cache misses or fetch breaks.
2. During dpred-mode, the processor does not encounter another diverge branch or a branch misprediction.
3. When the two predicated paths of a diverge branch do not merge, half of the fetched instructions are useful. This is not always true because the processor may reach the CFM point on one path. In that case, the processor would fetch instructions only from the path that did not reach the CFM point, which may or may not be the useful path.
4. The overhead of the select-μops is not included in the model. We found that this overhead is negligible; on average less than 0.5 fetch cycles per entry into dpred-mode.

Especially the first three assumptions do not always hold and therefore limit the accuracy of the model. However, accurate modeling of these limitations requires fine-grain microarchitecture-dependent, application-dependent, and dynamic-event-dependent information to be incorporated into the model, which would significantly complicate the model.

5. Diverge Loop Branches

DMP dynamically predicates low-confidence loop-type diverge branches to reduce the branch misprediction penalty in loops. If a mispredicted forward (i.e. non-loop) branch is successfully dynamically predicated, performance will likely improve. However, this is not necessarily true for loop branches. With dynamically-predicated loop branches, there are three misprediction cases (early-exit, late-exit and no-exit; similarly to wish loops [13]). Only the late-exit case provides performance benefit (see below). Hence, the cost-benefit analysis of loops needs to consider these different misprediction cases. In this section, we provide a cost-benefit model for the dynamic predication of diverge loop branches and describe simple heuristics to select diverge loop branches.

5.1. Cost-Benefit Analysis of Loops

The overhead of correctly-predicted case: Entering dpred-mode when a diverge loop branch is correctly predicted has performance overhead due to the select-μops inserted after each dynamically-predicated iteration. We model the cost of select-μops based on the number of fetch cycles they consume as shown below:

$$dpred\_overhead = N(select\_uops) * dpred\_iter/fw \quad (18)$$

$N(select\_uops)$: The number of select-μops inserted after each iteration
$dpred\_iter$: The number of loop iterations during dpred-mode

Misprediction case 1 (Early-exit): During dpred-mode, if the loop is iterated fewer times than it should be, the processor needs to execute the loop at least one more time, so it flushes its pipeline. Hence, the early-exit case has only the overhead of select-μops and no performance benefit. The overhead is calculated the same way as in the correctly predicted case (Equation (18)).

Misprediction case 2 (Late-exit): During dpred-mode, if the loop is iterated a few times more than it should be, the misprediction case

is called late-exit. Late exit is the only case for which the dynamic predication of a loop branch provides performance benefit because the processor is able to fetch useful control-independent instructions after the loop exit. In this case, the overhead is due to the cost of select-μops and extra loop iterations (that will become NOPs). However, instructions fetched after the processor exits the loop are useful and therefore not included in the overhead. The overhead of the late-exit case is thus calculated as follows:

$$dpred\_overhead = N(loop\_body) * dpred\_extra\_iter/fw + N(select\_uops) * dpred\_iter/fw \quad (19)$$

$N(loop\_body)$: The number of instructions in the loop body
$dpred\_extra\_iter$: The number of extra loop iterations in dpred-mode

Misprediction case 3 (No-exit): If the processor has not exited a dynamically-predicated loop until the loop branch is resolved, the processor flushes the pipeline just like in the case of a normal loop branch misprediction. Hence, the no-exit case has only overhead, which is the cost of select-μops as calculated in Equation (18).

Thus, the total cost of dynamically predicating a loop is:

$$\begin{aligned} dpred\_cost = {} & dpred\_overhead(corr\_pred) * P(enter\_dpred\_corr\_pred) \\ & + dpred\_overhead(early\_exit) * P(early\_exit) \\ & + dpred\_overhead(late\_exit) * P(late\_exit) \\ & + dpred\_overhead(no\_exit) * P(no\_exit) \\ & - misp\_penalty * P(late\_exit) \end{aligned} \quad (20)$$

$dpred\_overhead(X)$: dpred_overhead of case X

5.2. Heuristics to Select Diverge Loop Branches

According to the cost-benefit model presented in Section 5.1, the cost of a diverge loop branch increases with (1) the number of instructions in the loop body, (2) the number of select-μops (We found this is strongly correlated with the loop body size), (3) the average number of dynamically-predicated loop iterations ($dpred\_iter$), (4) the average number of extra loop iterations ($dpred\_extra\_iter$) in the late-exit case, and (5) the probability of a dynamic predication case other than late-exit. Unfortunately, a detailed cost-benefit analysis of each dynamic predication case requires the collection of per-branch profiling data obtained by emulating the behavior of a DMP processor. In particular, determining the probability of each misprediction case, the number of dynamically predicated iterations, and the number of extra iterations in the late-exit case requires either profiling on a DMP processor (with specialized hardware support for profiling) or emulating a DMP processor's behavior in the profiler. Since such a profiling scheme is impractical due to its cost, we use simple heuristics that take into account the insights developed in the cost-benefit model to select diverge loop branches. These heuristics do not select a loop branch as a diverge branch if any of the following is true:

1. If the number of instructions in the loop body is greater than $STATIC\_LOOP\_SIZE$.
2. If the average number of executed instructions from the loop entrance to the loop exit (i.e. the average number of instructions in the loop body times the average loop iteration count) based on profile data is greater than $DYNAMIC\_LOOP\_SIZE$. We found that there is a strong correlation between the average number of loop iterations and $dpred\_extra\_iter$. Hence, this heuristic filters branches with relatively high $dpred\_overhead$ for the late-exit case based on Equation (19).
3. If the average number of loop iterations (obtained through profiling) is greater than $LOOP\_ITER$. We found that when a branch has high average number of loop iterations, it has high $P(no\_exit)$.

In our experiments, we use $STATIC\_LOOP\_SIZE = 30$, $DYNAMIC\_LOOP\_SIZE = 80$, and $LOOP\_ITER = 15$, which we empirically determined to provide the best performance.

6. Methodology

6.1. Control-flow Analysis and Selection of Diverge Branch Candidates

We developed a binary analysis toolset to analyze the control-flow graphs, implement the selection algorithms presented in Section 3, and evaluate the diverge branch candidates using the cost-benefit model developed in Sections 4 and 5. The result of our analysis is a list of diverge branches and CFM points that is attached to the binary and passed to a cycle-accurate execution-driven performance simulator that implements a diverge-merge processor.

A limitation of our toolset is that the possible targets of indirect branches/calls are not available because our tool does not perform data flow analysis. Therefore, we cannot exploit possible diverge branches whose taken/not-taken paths encounter indirect branches/calls before reaching a CFM point. Implementing our techniques in an actual compiler can overcome this limitation because a compiler has source-level information about the targets of indirect branches/calls.

6.2. Simulation Methodology

We use an execution-driven simulator of a processor that implements the Alpha ISA. The parameters of the baseline processor and the additional support needed for DMP are shown in Table 1. The experiments are run using the 12 SPEC CPU2000 integer benchmarks and 5 SPEC 95 integer benchmarks.[7] Table 2 shows the relevant characteristics of the benchmarks. All binaries are compiled for the Alpha ISA with the -fast optimizations. The benchmarks are run to completion with a reduced input set [16] to reduce simulation time. Section 7.3 presents results obtained when the train input sets are used for profiling. All other sections present results with the reduced input set used for profiling.

7. Results

7.1. Diverge Branch Selection Algorithms

Figure 5 shows the performance improvement of DMP with different diverge branch selection algorithms. The left graph in Figure 5 shows the performance impact of adding the results of each selection algorithm one by one cumulatively: Alg-exact (exact), Alg-freq (exact+freq), short hammocks (exact+freq+short), return CFM points (exact+freq+short+ret), and loops (exact+freq+short+ret+loop).[8] All algorithms use thresholds that are empirically determined to provide the best performance.

According to Figure 5 (left) the performance benefit of DMP increases as we cumulatively employ our diverge branch selection tech-

[7] Gcc, vortex, and perl in SPEC 95 are not included because later versions of these benchmarks are included in SPEC CPU2000.

[8] exact+freq+short+ret+loop is called *All-best-heur* in the rest of the paper, standing for "all techniques, with the best empirically-determined thresholds, and using heuristics to select diverge branches."

Table 1. Baseline processor configuration and additional support needed for DMP

Front End	64KB, 2-way, 2-cycle I-cache; fetches up to 3 conditional not-taken branches
Branch Predictors	16KB (64-bit history, 256-entry) perceptron branch predictor [10]; 4K-entry BTB 64-entry return address stack; minimum branch misprediction penalty is 25 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
Memory System	L1 D-cache: 64KB, 4-way, 2-cycle, 2 ld/st ports L2 unified cache: 1MB, 8-way, 8 banks, 10-cycle, 1 port; All caches: LRU replacement and 64B line size 300-cycle minimum memory latency; 32 memory banks; bus latency: 40-cycle round-trip
DMP Support [12]	2KB (12-bit history, threshold 14) enhanced JRS confidence estimator [9, 6]; 32 predicate registers; 3 CFM registers

Table 2. Characteristics of the benchmarks: baseline IPC, mispredictions per kilo-instructions (MPKI), number of retired instructions (Insts), number of all static branches (All br.), number of static diverge branches (Diverge br.), and average of number of CFM points per diverge branch (Avg. # CFM). Diverge branches and CFM points are selected based on *All-best-heur* evaluated in Section 7.

	gzip	vpr	gcc	mcf	crafty	parser	eon	perlbmk	gap	vortex	bzip2	twolf	compress	go	ijpeg	li	m88ksim
Base IPC	2.10	1.58	1.09	0.45	2.24	1.30	3.17	1.91	1.94	3.26	1.42	2.17	2.29	0.86	2.88	2.07	2.27
MPKI	5.1	9.4	12.6	5.4	5.5	8.3	1.7	3.6	1.0	1.0	7.7	6.0	5.2	23.0	4.5	5.9	1.3
Insts (M)	249	76	83	111	190	255	129	99	404	284	316	101	150	137	346	248	145
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
Diverge br.	175	272	2364	86	643	167	205	513	286	319	97	358	24	1286	117	21	136
Avg. # CFM	1.02	1.02	1.03	1	1.07	1.02	1.05	1.03	1.03	1.03	1.01	1.02	1.04	1.04	1.02	1	1.04

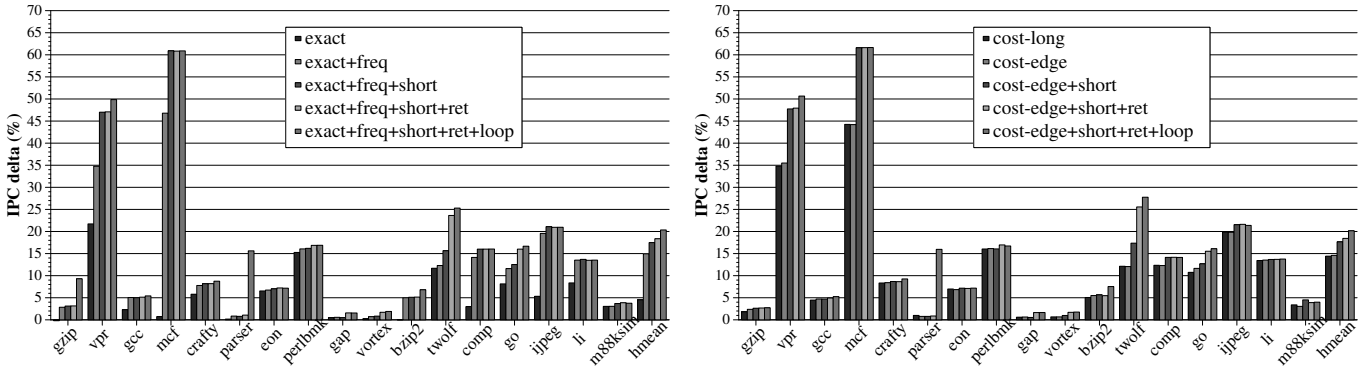

Figure 5. Performance improvement of DMP with different selection algorithms: (left) Alg-exact and Alg-freq (right) cost-benefit analysis

niques. Using just Alg-exact, DMP provides a performance improvement of 4.5%. However, when all our techniques are used, the performance improvement of DMP increases to 20.4%. Figure 6 provides insight into the performance increases by showing the number of pipeline flushes in the baseline processor and in DMP. As we employ more and more of the proposed branch selection algorithms, the number of pipeline flushes due to branch mispredictions decreases. These results demonstrate that the proposed mechanisms are effective at selecting diverge branches that provide performance benefits when dynamically predicated.

As shown in Figure 5 (left), selecting frequently-hammocks (Alg-freq) improves average performance by 10% on top of Alg-exact. Hence, the selection of frequently-hammocks is the largest contributor to the performance of dynamic predication. Always predicating short hammocks improves performance by 2.2% on average and by more than 4% in vpr (12%), mcf (14%) and twolf (4%). Vpr and twolf have many short hammocks that are highly mispredicted and, thus, always predicating them provides significant improvements. In mcf, the most highly mispredicted branch is a short hammock branch whose predication provides a 14% performance benefit. Including return CFM points improves performance by 0.8% on average and by

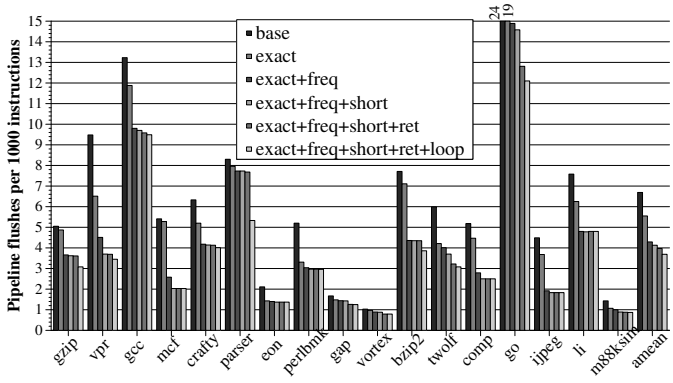

Figure 6. Pipeline flushes due to branch mispredictions in the baseline and DMP

more than 3% in twolf (8%) and go (3.5%). Twolf and go have many hammocks inside function calls that merge at different return instructions. Those hammocks cannot be diverge branches without the return CFM point mechanism. Finally, selecting diverge loop branches using the heuristics described in Section 5 provides an additional 1.7% av-

erage performance improvement, especially in gzip (6%) and parser (14%). Parser has a frequently-executed small loop in which an input word is compared to a word in the dictionary. The exit branch of this loop is frequently mispredicted (because the lengths of the input words are not predictable), and therefore its dynamic predication results in a large performance benefit.

The right graph in Figure 5 shows the performance improvement of DMP if we use the cost-benefit analysis developed in Section 4 to select diverge branches. The compiler uses two different methods to calculate the overhead of dynamic predication: longest path (cost-long), method 2 in Section 4.1.1, and edge-profile-based average path (cost-edge), method 3 in Section 4.1.1. The cost-edge method provides slightly higher performance than the cost-long method because cost-edge calculates the overhead of dynamic predication more precisely. Figure 5 (right) also shows the performance impact of adding each algorithm in sequence with the edge-profiling based cost-benefit analysis: always predicating short hammocks (cost-edge+short), return CFM points (cost-edge+short+ret), and diverge loops (cost-edge+short+ret+loop).[9] Using all these optimizations in conjunction with cost-edge results in 20.2% performance improvement over the baseline processor. Therefore, we conclude that using cost-benefit analysis (which does not require the optimization of any thresholds) to determine diverge branches can provide the same performance provided by using optimized threshold-based heuristics in conjunction with Alg-exact and Alg-freq.

7.1.1. Effect of Optimizing Branch Selection Thresholds Figure 7 shows the performance improvement for different MIN_MERGE_PROB and MAX_INSTR thresholds when the compiler uses only Alg-exact and Alg-freq. The results show that it is better to choose lower MIN_MERGE_PROB when the number of instructions between a diverge branch and the CFM is less than 50, since the overhead of entering dpred-mode for these small hammocks is relatively low. When MAX_INSTR is 100 or 200, MIN_MERGE_PROB=5% results in the best average performance. On average, MAX_INSTR=50, MAX_CBR=5, and MIN_MERGE_PROB=1% provides the best performance, so we used these thresholds for all other experiments that do not use the cost-benefit model to select diverge branches. Using a too small (e.g. 10) or too large (e.g. 200) threshold value for MAX_INSTR hurts performance. A too small MAX_INSTR value prevents many mispredicted relatively large hammocks from being dynamically predicated, thereby reducing the performance potential. A too large MAX_INSTR value causes the selection of very large hammocks that fill the instruction window in dpred-mode, which significantly reduces the benefit of dynamic predication.

Note that not selecting the best thresholds results in an average performance loss of as much as 3%. Therefore, optimizing the thresholds used in our heuristic-based selection algorithms is important to obtain the best performance. This observation also argues for the use of the analytical cost-benefit model that does not require the optimization of any thresholds to provide equivalent performance.

Another conclusion from Figure 7 is that selecting only those CFM points with a large merging probability ($MIN\_MERGE\_PROB = 90\%$) provides most of the performance benefit in DMP. Adding CFM point candidates with smaller merge probabilities incrementally improves average performance by at most 3%, but selecting candidates with a merge probability lower than 30% provides only negligible (less than 0.1%) benefit. Thus, DMP gains most of its performance from the frequently executed paths in which control-flow is very likely to merge at a control-independent point. This result can be used to optimize (i.e. reduce) the number of CFM points supported by the DMP ISA, but a thorough discussion of the tradeoffs in the DMP ISA is out of the scope of this paper.

[9] cost-edge+short+ret+loop is called *All-best-cost* in the rest of the paper.

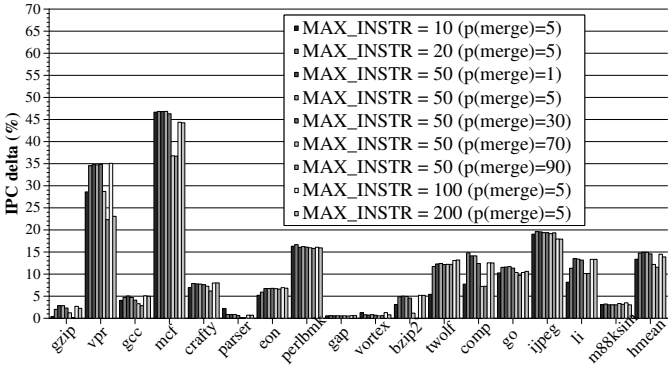

Figure 7. Performance improvement of DMP with different MAX_INSTR and MIN_MERGE_PROB heuristics

7.2. Comparisons with Other Diverge Branch Selection Algorithms

Since there is no previous work on compilation for DMP processors, we compare our algorithms with several simple algorithms to select diverge branches. Figure 8 compares the performance of six different algorithms: (1) *Every-br*: This is the extreme case where all branches in the program are selected as diverge branches, (2) *Random-50*: 50% of all branches are randomly selected, (3) *High-BP-5*: All branches that have higher than 5% misprediction rate during the profiling run are selected, (4) *Immediate*: All branches that have an IPOSDOM are selected. (5) *If-else*: Only if and if-else branches with no intervening control-flow are selected, (6) *All-best-heur*: Our best-performing algorithm. Note that for the simple algorithms (1), (2) and (3), not all branches have corresponding CFM points.[10] If there is no CFM point for a low-confidence diverge branch, then the processor stays in dpred-mode until the branch is resolved, and any performance benefit would come from dual-path execution.

Figure 8 shows that *Every-br*, *High-BP-5*, and *Immediate* are the best-performing simple algorithms for selecting diverge branches with average performance improvements of 4.4%, 4.3% 4.5% respectively. However, none of these other algorithms provide as large performance improvements as our technique, which improves average performance by 20.4%. We conclude that our algorithms are very effective at identifying good diverge branch candidates.

Note that *Every-br*, *High-BP-5*, and *Immediate* show relatively large performance improvements in benchmarks where a large percentage of the mispredicted branches are simple hammock branches (e.g. eon, perlbmk, and li). Only in gcc does one simple algorithm

[10] If a branch has an IPOSDOM, the IPOSDOM is selected as the CFM point in the explored simple algorithms.

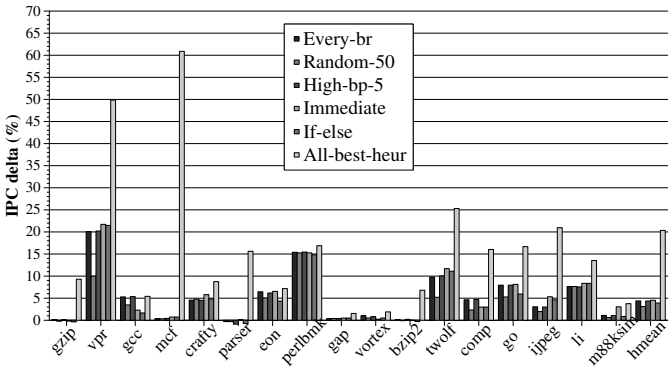

Figure 8. Performance improvement of DMP with alternative simple algorithms for selecting diverge branches

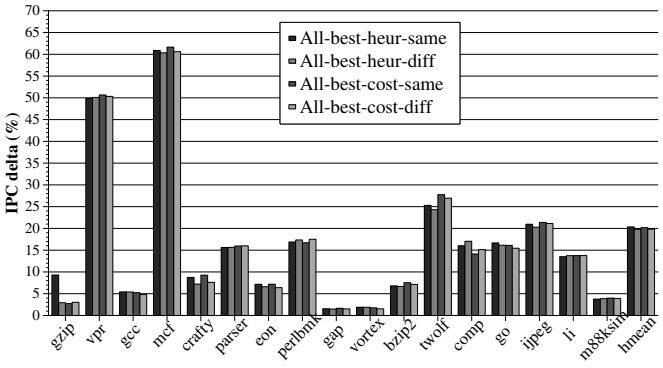

Figure 9. Performance improvement of DMP when a different input set is used for profiling

(*Every-br*) perform almost as well as our scheme. Gcc has very complex CFGs (that usually do not result in frequently-hammocks), so there are few diverge branch candidates. Gcc also has a very high branch misprediction rate (7%). *Every-br* allows the processor to enter dpred-mode for *all low-confidence branches*, which covers 62% of all mispredicted branches. Therefore, *Every-br* provides a similar performance improvement as that of entering dpred-mode for only carefully selected branches, which covers only 30% of all mispredicted branches.

7.3. Input Set Effects

We developed the algorithms and heuristics in previous sections by profiling and evaluating with the same input set to exclude the effects of input-set variations on the evaluation. In this experiment, we use the same algorithms and the same heuristic values developed in the previous sections, but we profile with the *train* input set to select diverge branches and CFM points. Figure 9 shows the DMP performance when the profiling input set is the same as the run-time input set (*same*) versus when the profiling input set is different from the run-time input set (*diff*). The compiler uses the best performing heuristic-based optimizations (*All-best-heur-same*, *All-best-heur-diff*) and the cost-benefit model with all optimizations (*All-best-cost-same*, *All-best-cost-diff*).

Figure 9 shows that the performance improvement provided by DMP is 19.8% (both *All-best-heur-diff* and *All-best-cost-diff*) when different input sets are used for profiling and actual runs. These improvements are only very slightly (0.5%) lower than when the same input set is used for profiling and actual runs. Only in gzip does profiling with the same input set significantly outperform profiling with a different input set (by 6%) when the compiler uses *All-best-heur* to select diverge branches. Hence, we find that DMP performance is not significantly sensitive to differences in the profile-time and run-time input sets.

Figure 10 shows whether or not the compiler finds the same set of diverge branches across input sets. We classify diverge branches into three groups: (1) *Only-run*: branches that are selected only when the compiler uses the run-time input set (MinneSPEC's reduced input set [16]) for profiling, (2) *Only-train*: branches that are selected only when the compiler uses a different input set (SPEC's train input set) for profiling, (3) *Either-run-train*: branches that are selected when the compiler uses either input set for profiling. The bars in Figure 10 show

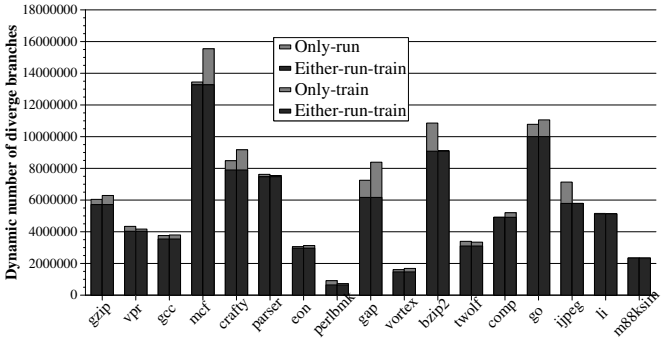

Figure 10. Dynamic diverge branches selected by different input sets (only run-time, only train, or either input). Left bar: profiling with run-time input, Right bar: profiling with train input

the classification of diverge branches when respectively the run-time (left) and train (right) input sets are used for profiling.

More than 74% of all dynamic diverge branches in all benchmarks are selected when either input set is used for profiling. Thus, most of the diverge branches identified by profiling with different input sets are the same. Only gap (26%) has more than 20% and mcf (14%), crafty (13%), vortex (13%), bzip2 (16%) and ijpeg (18%) have more than 10% of all dynamic diverge branches that are classifed as either *only-run* or *only-train*. However, even with differences of 10-20% in the dynamic diverge branches selected by profiling with different input sets, only mcf (1%) and crafty (1.6%) show more than 1% IPC degradation when a different input set is used for profiling. This is due to two major reasons: (1) programs have similar sets of highly mispredicted static branches across different input sets [3], (2) even though a branch may be marked as a diverge branch by the compiler, only low-confidence diverge branches are actually predicated at run-time; therefore the selection of a slightly different set of branches with different profiling input sets does not necessarily mean that the set of dynamically predicated branches will be significantly different.

We can make the following conclusions based on our results:

1. Our diverge branch selection algorithms are not significantly sensitive to differences in the profiling input set.
2. The dynamic nature of predication in the DMP architecture mitigates the effects of changing the profiling input set by selectively entering dpred-mode and dynamically choosing which CFM points to use at run-time.

8. Related Work

8.1. Branch Selection for Dynamic Predication

The most relevant work to ours is Klauser et al. [15], which briefly describes how they select branches for a processor that can dynamically predicate very simple control flow hammocks (i.e. hammocks with no intervening control-flow inside). They used two compile-time methods, size-based selection and profile-based selection, to select branch candidates for dynamic predication. The size-based method uses the number of instructions between a branch and the join point (i.e. CFM-point in DMP) of the branch to select suitable simple hammocks. The profile-based method uses a cost-benefit model, similar to our cost-benefit model but applicable to only simple hammocks. Their cost-benefit model took into account the branch misprediction rates, but they did not consider the accuracy of the confidence estimator. Our work includes both size-based heuristics and cost-benefit analysis using profile data. Our compiler algorithms provide more generalized selection mechanisms and cost-benefit analysis for not only simple hammocks, but also more complex control-flow structures (nested hammocks, frequently-hammocks, and loops) to support DMP.

8.2. Branch Selection for Static Predication

Less relevant to our work is the body of literature on branch selection algorithms for static if-conversion [1, 19] for predicated instruction set architectures. Static predicated code generation algorithms use edge profiling and/or the number of instructions in a region that is considered for static predication to decide whether or not to if-convert a branch instruction. Both Pnevmatikatos and Sohi [20] and Tyson [23] used the number of instructions in a region to determine whether a short forward branch should be if-converted. Chang et al. converted highly mispredicted branches to predicated code [3].

Mantripragada and Nicolau [18] developed compiler algorithms to select static if-conversion candidates based on basic block sizes (in terms of the number of instructions) and branch misprediction profile data. Our cost-benefit model presented in Section 4 is conceptually similar to Mantripragada and Nicolau's cost-benefit model for static predication in that both models try to select branches for which predication would provide performance benefits. However, as dynamic predication is different from static predication, we consider dynamic effects such as the accuracy of the confidence estimator and merge probability. Furthermore, we provide a new analytical model to select candidates for frequently-hammocks and loops, which cannot be predicated by conventional if-conversion.

Hyperblock formation [17] uses path execution frequencies, basic block sizes, and basic block characteristics to decide which blocks should be included in a hyperblock. Hyperblocks enhance the compiler's scope for code optimization by increasing basic block sizes. Hence, identifying hot-paths is more important than identifying highly mispredicted branches. August et al. [2] proposed a framework that considers branch misprediction rate and instruction scheduling effects due to predication in an EPIC processor to decide which branches would not benefit from if-conversion and should be reverse if-converted [25]. They also proposed a cost-benefit model for statically predicated code.

8.3. Input Set Differences in Profiling for Predication

Chang et al. [3] compared the set of frequently mispredicted branches between different input sets of SPEC 95 applications. They found that if an input set resulted in the execution of most of the static branches in an application, then the set of frequently mispredicted branches (i.e. branches that are if-conversion candidates in their if-conversion algorithms) would be similar across different input sets.

Sias et al. [22, 21] evaluated the performance variation in hyperblock-based code optimizations due to variations in the input set used for profiling. They found that among SPEC CPU2000 benchmarks, crafty, perlbmk and gap showed more than 3% performance difference when profiled with the train input set versus the reference input set. In hyperblocks, the processor does not have the ability to change the statically optimized code. Thus, if the profiling input set is not similar to the run-time input set, there can be significant performance variations as the hardware cannot override a possibly wrong decision made by the compiler based on the profiling input set. In contrast to hyperblocks (and static predication), DMP has the ability to dynamically choose which dynamic instances of each branch to predicate. Hence, DMP is less sensitive to differences between input sets used for profiling and actual execution.

Hazelwood and Conte [7] discussed the performance problems associated with statically predicated code when the input set of the program changes. They used dynamic profiling to identify hard-to-predict branches and dynamically re-optimized the code based on the run-time behavior of branches. Both software-based dynamic optimization and hardware-based dynamic predication reduce the dependence of predication performance on the profile-time input set.

Kim et al. [14] evaluated the extent of variations in branch misprediction rate across input sets. They proposed a profiling algorithm (2D-profiling) that can detect input-dependent branches (i.e. branches whose misprediction rates change significantly across input sets) using a single profiling run. Our work can be further improved by incorporating the 2D-profiling scheme to our algorithms to select only possibly mispredicted branches as diverge branches. Excluding *always easy-to-predict branches* from selection as diverge branches would reduce the static code size and also reduce the potential for aliasing in the confidence estimator.

9. Conclusion and Future Work

This paper presented and evaluated new code generation algorithms for dynamic predication in the diverge-merge processor (DMP) architecture. The proposed algorithms select branches that are suitable and profitable for dynamic predication based on profiling information. We explored diverse heuristics to select hammock and loop diverge branches and corresponding control-flow merge (CFM) points, and some optimizations based on program characteristics: always-predicating short hammocks and return CFM points. We also proposed a new profile-driven analytical cost-benefit model to select branches that are profitable for dynamic predication.

Our results show that, with the proposed branch selection algorithms, a DMP processor outperforms an aggressive baseline processor by 20.4%. In contrast, the best-performing alternative branch selection algorithm results in a performance increase of only 4.5% over the baseline.

Our future work includes the exploration of more accurate cost-benefit models. In particular, the proposed cost model for loop diverge branches requires the profiler to collect DMP-specific information. We intend to examine techniques that can make the cost model for

selecting loop branches implementable. Exploration of dynamic profiling mechanisms that collect feedback on the usefulness of dynamic predication at run-time and accordingly enable/disable dynamic predication is another promising avenue for future research.

Acknowledgments

We thank Robert Cohn, Moinuddin Qureshi, Aater Suleman, Mark Oskin, members of the HPS research group, and the anonymous reviewers for their 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] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *POPL-10*, 1983.

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

[3] 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.

[4] K. D. Cooper, T. J. Harvey, and K. Kennedy. A simple, fast dominance algorithm. *Software Practice and Experience*, 4:1–10, 2001.

[5] 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. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.

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

[7] K. Hazelwood and T. Conte. A lightweight algorithm for dynamic if-conversion during dynamic optimization. In *PACT*, 2000.

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

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

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

[11] R. E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, 1999.

[12] H. Kim, J. A. Joao, O. Mutlu, and Y. N. Patt. Diverge-merge processor (DMP): Dynamic predicated execution of complex control-flow graphs based on frequently executed paths. In *MICRO-39*, 2006.

[13] 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.

[14] H. Kim, M. A. Suleman, O. Mutlu, and Y. N. Patt. 2D-profiling: Detecting input-dependent branches with a single input data set. In *CGO-4*, 2006.

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

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

[17] 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*, pages 45–54, 1992.

[18] S. Mantripragada and A. Nicolau. Using profiling to reduce branch misprediction costs on a dynamically scheduled processor. In *ICS*, 2000.

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

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

[21] J. W. Sias. *A Systematic Approach to Delivering Instruction-Level Parallelism in EPIC Systems*. PhD thesis, University of Illinois at Urbana-Champaign, June 2005.

[22] 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.

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

[24] 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.

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