Adaptive Predication via Compiler-Microarchitecture Cooperation

Hyesoon Kim

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

TR-HPS-2007-003
August 2007

This page is intentionally left blank.

The Dissertation Committee for Hyesoon Kim
certifies that this is the approved version of the following dissertation:

Adaptive Predication via Compiler-Microarchitecture Cooperation

Committee:

Yale N. Patt, Supervisor

Craig M. Chase

Steve Keckler

Derek Chiou

Jared Stark

Adaptive Predication via Compiler-Microarchitecture Cooperation

by

Hyesoon Kim, B.S.; M.S.; M.S.E.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2007

Dedicated to my parents

Acknowledgments

After a long formal education journey, I finally have a chance to thank everybody who taught me, helped me, and shared their friendship with me during my Ph.D. program. This dissertation cannot be completed without support from them, especially my parents Sun Sam Kim, Sook Ja Lee and my older brother Phill Soon Kim. Here I am trying to express my gratitude to everybody although these words are never enough.

First of all, I thank my adviser, Yale N. Patt, who taught me how computers work and also showed me what I really like. His passion during EE360N helped me find what I want to study for my Ph.D. He invited me to join the HPS research group when I was barely able to speak English and when I did not have enough background in the computer engineering field. And then, he encouraged me to solve important and difficult problems after building a strong background.

I thank every member of the HPS group. I have felt lucky to be a member of such a wonderful research group. I especially thank Onur Mutlu. My Ph.D. life would have been very different without him. He has always open to strong technical discussions and showed me how to write strong papers. This dissertation became much stronger and clearer with his contributions.

I thank Francis Tseng for always being helpful in not only technical support but also helping me with various kinds of user errors. I thank Moinuddin Qureshi for many productive and joyful coffee meetings especially during our last year of graduate school.

I had a great time working with José A. Joao, especially on the topic of DMP. I thank him for the discussions, support, and the chance to work with him. I also thank M. Aater Suleman for working on 2D-profiling together with me. I thank Chang Joo Lee for useful discussions and many different kinds of support including frequent rides. I thank Venyu Narasiman for providing helpful comments for many paper drafts, which eventually became parts of my dissertation.

I thank David N. Armstrong for his help in developing the IA-64 simulator, for educating me in the ways of the American culture, and for being a friend. I thank Santhosh Srinath for answering my technical questions and accompanying me in many hiking trips in Oregon and Texas. I thank David Thompson and Linda Hastings for their comments and suggestions in many paper drafts, and Danny Lynch and Rustam Miftakhutdinov for providing a cheerful office environment. I thank the senior members of HPS, Mary Brown, Robert Chappell, Paul Racunas, and Sangwook Peter Kim, for being accessible mentors and for their contributions to our group's simulation infrastructure.

I would also like to thank Craig Chase, Derek Chiou, Steve Keckler and Jared Stark for taking the time to serve on my dissertation committee. Jared Stark provided the initial idea of wish branches, which is part of this dissertation. Derek Chiou was always open to discussions and always gave me kindly advice.

I learned a lot during my internships in industry. Eric Sprangle and Jared Stark walked me through the first steps of being a computer architect. I especially thank Robert Cohn giving me a chance to intern with the VSSAD group and for supporting Pin. Konrad Lai, Srikanth Srinivasan, Roy Ju, Joel Emer, Jon Pieper, and Sanjay Patel provided valuable comments on my research.

All my friends, Eunmi Park, Hyunjung Lee, Yoonjung Hong, many ECE folks including Dam Sunwoo, Joonsoo Kim and many other friends helped me directly or indirectly during my Ph.D. studies. Once again, I gratefully thank everybody and thank my parents.

Hyesoon Kim
July 2007, Austin, Texas

Adaptive Predication via Compiler-Microarchitecture Cooperation

Publication No. ____________________

Hyesoon Kim, Ph.D.
The University of Texas at Austin, 2007

Supervisor: Yale N. Patt

Even after decades of research in branch prediction, branch predictors still remain imperfect, which results in significant performance loss in aggressive processors that support large instruction windows and deep pipelines. Predicated execution can reduce the number of branch mispredictions by eliminating hard-to-predict branches. However, the additional instruction overhead and data dependencies due to predicated execution sometimes offset the performance benefits of having fewer mispredictions. This dissertation presents two cooperative compiler-microarchitecture mechanisms to reduce the branch misprediction penalty by combining predicated execution and branch prediction.

The first mechanism is a set of new control flow instructions, called wish branches. With wish branches, the compiler generates code that can be executed either as normal branch code or as predicated code. At run-time, the hardware chooses between normal branch code and predicated code based on the run-time branch behavior and the estimated run-time effectiveness of each solution. The results show that wish branches can significantly improve both performance and energy efficiency compared to predication or branch prediction.

To provide the benefit of predicated code to non-predicated Instruction Set Architectures (ISAs) and to increase the benefit of predicated execution beyond the benefit of wish branches, this dissertation also presents and evaluates the Diverge-Merge Processor (DMP) architecture. In the diverge-merge processor, the compiler analyzes the control-flow graphs of the program and marks branches suitable for dynamic predication –called diverge branches– and their corresponding control flow merge points. The hardware not only chooses whether to use branch prediction or predication, but also decides “which” instructions after a branch should be predicated based on run-time branch behavior. This solution significantly reduces the overhead of predicated code and allows a very large set of control-flow graphs to be predicated, neither of which was possible previously because predication was performed statically without any run-time information. This dissertation compares DMP with all other major previously-proposed branch processing paradigms available in the literature in terms of performance, power, energy consumption, and complexity. The results show that DMP is the most energy-efficient and high-performance paradigm for branch handling. Code generation algorithms for the DMP architecture and cost-benefit analysis models of dynamic predication are also evaluated.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Today's high performance processors employ deep pipelines to support high clock frequencies. Some processing cores in near-future chip multiprocessors are expected to support a large number of in-flight instructions [56, 17, 14, 75, 23] to extract both memory-level parallelism (MLP) and instruction level parallelism (ILP) in order to obtain high performance and energy-efficiency on the serial portions of applications [55]. The performance benefit and energy efficiency of both pipelining and supporting a large number of in-flight instructions depend critically on the accuracy of the processor's branch predictor [73, 56, 75]. Even after decades of research in branch prediction, branch predictors still remain imperfect. Hard-to-predict branches are frequently mispredicted, and they not only limit performance but also result in wasted energy.

1.1 The Problem: The Limitations of Predicated Execution

Predication has been used to avoid pipeline flushes due to branch mispredictions by converting control dependencies into data dependencies [3]. 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. **Adaptivity:** Predication is not adaptive to run-time branch behavior because a statically if-converted branch instruction remains if-converted regardless of whether or

not its instances are hard-to-predict at run-time.

Figure 1.1 shows the execution time of predicated code binaries with different inputs. The data is measured on an Itanium-II machine and binaries are compiled with the ORC-2.0 compiler [57]. Data is normalized to the execution time of a non-predicated code binary for each input. The results show that predicated code binaries generally provide performance benefit over the non-predicated code binaries. But, they sometimes perform worse. For example, for mcf, predicated code provides a 9% performance improvement for input-C, but causes a 4% performance loss for input-A. For bzip2, predicated code only provides a 1% improvement for input-C, but causes a 16% loss for input-A. Hence, the performance of predicated execution is highly dependent on the run-time input set of the program. Appendix A will discuss input dependent branches.

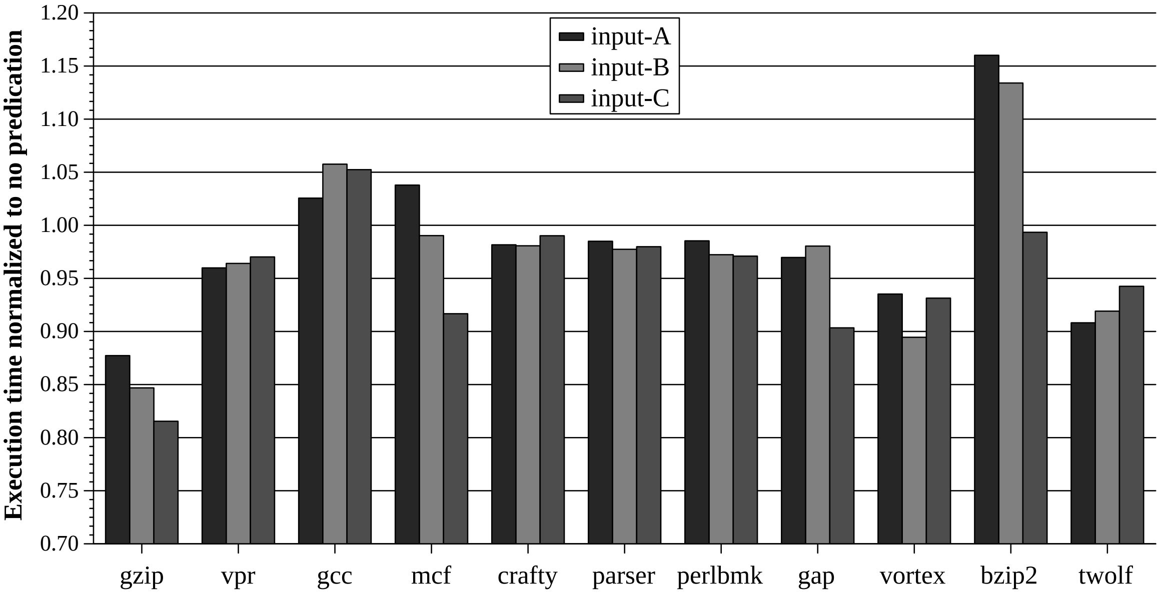


Figure 1.1: Relative execution time normalized to a non-predicated binary on a real Itanium-II processor.

2. **Complex CFG:** The performance potential of predication is limited because a large set of control-flow graphs (CFGs) either cannot be or are usually not converted to predicated code by compilers because they are too complex or they contain loops, function calls, indirect branches, too many instructions [16, 3, 57, 29]. Current compilers [57, 29] usually do not predicate large and complex CFGs because their predication would cause a large performance overhead.
3. **Instruction Set Architecture (ISA):** Predication requires significant changes to the ISA, in particular the addition of predicate registers and predicated instructions.

To overcome these three limitations/problems, this dissertation proposes and evaluates *adaptive predicated execution*.

1.2 Solution: Adaptive Predicated Execution

The adaptive predicated execution paradigm provides a choice to the hardware: the choice of whether or not to use predicated execution for *each dynamic instance* of a branch instruction. The compiler is not good at deciding which branches are hard-to-predict because it does not have access to run-time information. In contrast, the hardware has access to accurate run-time information about each branch. The adaptive predicated execution paradigm divides the work of predication between the hardware and the compiler based on what each of them is better at: the compiler is better at analyzing the control flow comprehensively and generating code and the hardware is better at making decisions based on observed run-time behavior. With adaptive predicated execution, the hardware can efficiently choose between predication and branch prediction depending on whether the branch is hard-to-predict or easy-to-predict.

The adaptive predicated execution paradigm includes two mechanisms: *wish branches* and the *diverge-merge processor (DMP)* architecture.

With wish branches, the compiler produces code that can be executed either as predicated code or normal branch code. At run-time the hardware can efficiently choose between predicated code and conditional branch code depending on whether the branch is hard-to-predict or easy-to-predict.

Wish branches can overcome the lack of adaptivity problem but inherit the limitations of software predication (the ISA problem and the complex CFG problem) except they can be applied to loop branches. To overcome all three problems, the diverge-merge processor (DMP) is proposed. In DMP, in contrast to the wish branch mechanism, the compiler does not produce a predicated version of the code, but it provides control-flow information to simplify the hardware used for dynamically predicating the code. The compiler marks suitable branches in the binary as candidates for dynamic predication. These branches are called *diverge branches*. The compiler also marks the control-flow merge point corresponding to each diverge branch. If a diverge branch is hard-to-predict at run-time, the processor dynamically predicates the instructions between the diverge branch and the control-flow merge point using the hints provided by the compiler. Hence, hard-to-predict branches can be eliminated at run-time through cooperation between the compiler and microarchitecture without requiring full support for predication (i.e., predicate registers and predicated instructions) in the ISA.

1.3 Thesis Statement

Adaptive predicated execution is a generalized and energy-efficient compiler and microarchitecture cooperation technique that can reduce the branch misprediction penalty in high performance processors.

1.4 Contributions

- **Branch instruction handling**: This dissertation presents both wish branches and the diverge-merge processor, which are two new techniques for reducing the branch misprediction penalty by combining the benefits of predicated execution and branch prediction in energy-efficient ways that do not significantly increase the hardware complexity.

- **Overcoming the limitations of predicated execution**: This dissertation presents the diverge-merge processor (DMP) architecture to overcome the major limitations/problems of software predication: adaptivity, complex-CFG, and ISA problems. This dissertation also presents profile-driven compiler code generation algorithms for dynamic predicated execution in the DMP architecture.

- **Predication of backward branches**: This dissertation presents wish loops that exploit predicated execution to reduce the branch misprediction penalty for backward (loop) branches.

1.5 Dissertation Organization

This dissertation is organized into seven chapters. Chapter 2 provides background on predicated execution. Chapter 3 shows related work in branch handling paradigms. Chapter 4 presents wish branches, and evaluates the performance benefit of wish branches. Chapter 5 presents and evaluates the diverge-merge processor architecture, which overcomes the three major limitations of predicated execution. Chapter 6 discusses and evaluates compiler algorithms for the diverge-merge processor. Finally, Chapter 7 provides conclusions, a summary of the key results and insights presented in this dissertation, and future directions for adaptive predicated execution.

Chapter 2

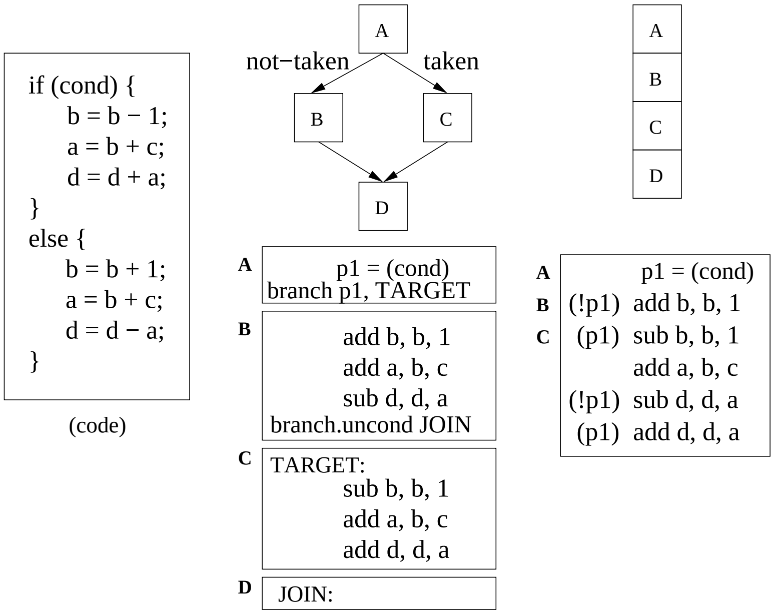
Background on Predicated Execution

This chapter provides a brief background on predicated execution. The next chapter will describe the previous research on predicated execution.

2.1 Predicated Execution

Figure 2.1 shows an example source code, the corresponding assembly code with branches (normal branch code - 2.1a) and the corresponding assembly code with predication (predicated code - 2.1b). In branch prediction, the processor speculatively fetches and executes block B or C based on the predicted direction of the branch in block A. When the branch is mispredicted, the processor flushes its pipeline and rolls back to the end of block A and fetches the alternate block. In predication, the processor fetches both block B and block C. Instructions in blocks B and C are not executed until the predicate value (p1 in Figure 2.1) is resolved.[1] Since there is no speculative execution, there is no pipeline flush penalty. However, the processor always fetches and executes instructions from both control-flow paths.

[1] Depending on microarchitecture designs, predicated instructions can be executed first and later committed when the predicate value is evaluated (e.g., the predicate slip mechanism in [79]). However, even in that design, the instructions can be executed first but the results of the predicated instructions still cannot be used by the later instructions until the predicate value is ready.



```
if (cond) {
      b = b − 1;
      a = b + c;
      d = d + a;
}
else {
      b = b + 1;
      a = b + c;
      d = d − a;
}
```

(code)

```
A        p1 = (cond)
     branch p1, TARGET
B           add b, b, 1
            add a, b, c
            sub d, d, a
     branch.uncond JOIN
C    TARGET:
            sub b, b, 1
            add a, b, c
            add d, d, a
D    JOIN:
```

(a)

```
A           p1 = (cond)
B  (!p1)  add b, b, 1
C   (p1)  sub b, b, 1
          add a, b, c
   (!p1)  sub d, d, a
    (p1)  add d, d, a
```

(b)

Figure 2.1: Source code and the corresponding assembly code for (a) normal branch code (b) predicated code

2.1.1 The Cost of Predicated Execution

Equations (2.1) and (2.2) show the cost of normal branch code and the cost of predicated code respectively. The compiler decides whether a branch is converted into predicated code or stays as a branch based on Equation (2.3) [57, 53].

$$\begin{aligned} Exec\_cost(normal\ branch) &= exec\_T * P(T) + exec\_N * P(N) \quad (2.1)\\ &\quad +misp\_penalty * P(misp) \\ Exec\_cost(predicated\ code) &= exec\_pred \quad (2.2)\\ Exec\_cost(normal\ branch) &> Exec\_cost(predicated\ code) \quad (2.3)\end{aligned}$$

$exec\_T$: Execution time of the code when the branch under consideration is taken,

$exec\_N$: Execution time of the code when the branch under consideration is not taken,

$P(case)$: The probability of the case; e.g., P(T) is the probability that the branch is taken,

$misp\_penalty$: Machine-specific branch misprediction penalty, and

$exec\_pred$: Execution time of the predicated code.

To demonstrate how sensitive Equation (2.1) is to the branch misprediction rate, we apply the equation to the code example shown in Figure 2.1. We set misp_penalty to 30 cycles, exec_T to 3 cycles, exec_N to 3 cycles, exec_pred to 5 cycles. Figure 2.2 displays the two equations, (2.1) and (2.2), as the branch misprediction rate is varied on the X-axis. With the given parameters, if the branch misprediction rate is less than 7%, normal branch code takes fewer cycles to execute than predicated code. If the branch misprediction rate is greater than 7%, predicated code takes fewer cycles than normal branch code. Therefore, we need a mechanism which chooses between branch prediction and predication depending on the run-time branch behavior.

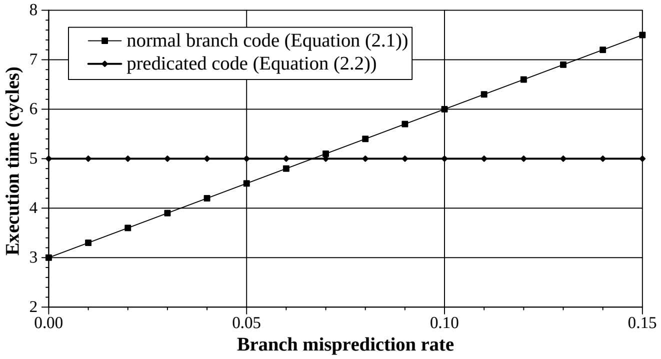


Figure 2.2: Execution time of predicated code and non-predicated code vs. branch misprediction rate

2.1.2 Nested Hammocks

Predicated execution can be used not only for a simple hammock (i.e., `if-else`) but also for a nested hammock (i.e., nested `if-else`). Figure 2.3 shows an example of branch code and predicated code for a nested hammock. In this example, the branch at basic block B, which is the control-flow dependent branch, should be executed only if p1 value is TRUE (i.e., when the branch at basic block A is taken). `AND` operation is performed over the predicate values to change p2 value only if p1 value is TRUE.

2.2 Microarchitecture Support for Out-Of-Order Processors: Register Renaming Problem

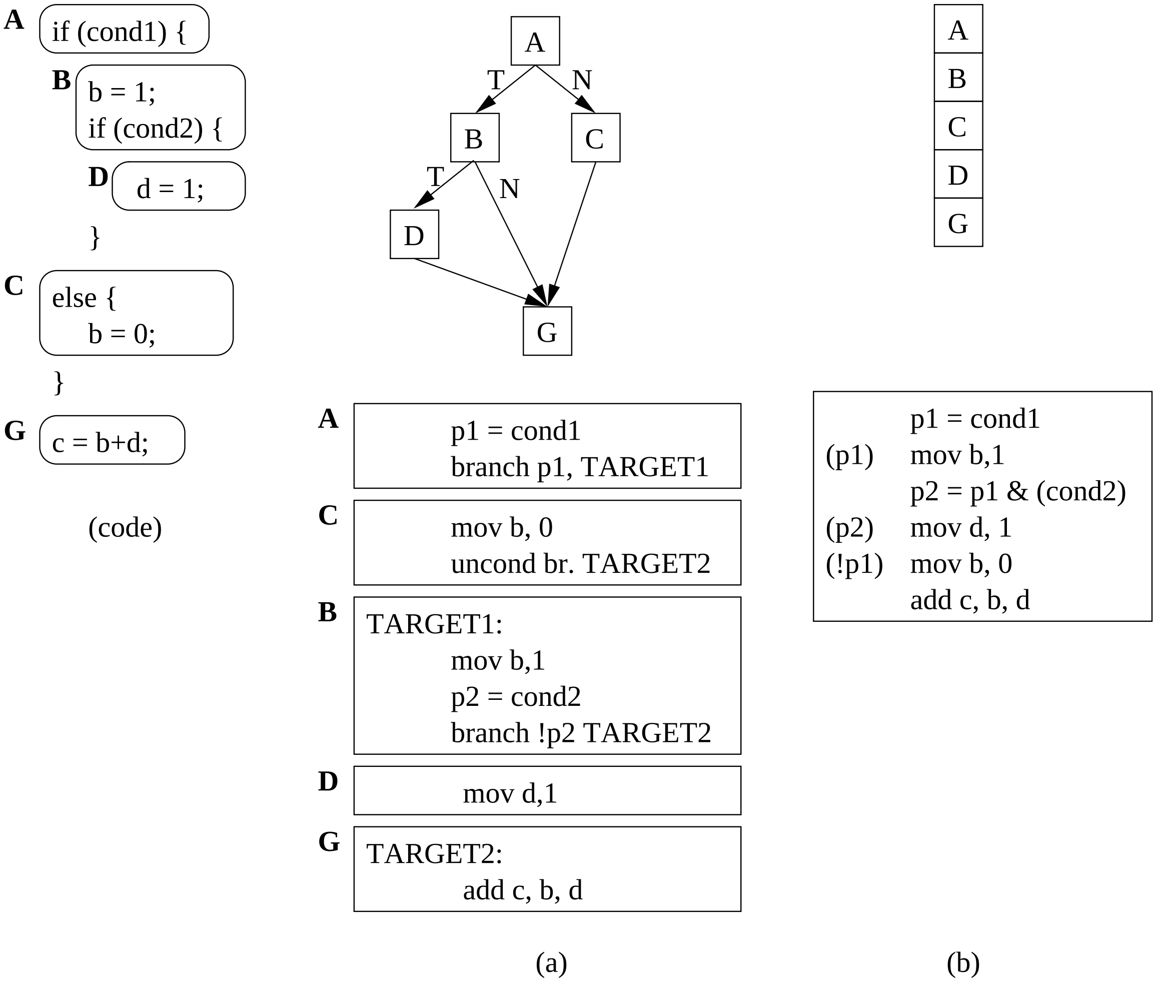
In an out-of-order processor, predication complicates register renaming because a predicated instruction may or may not write into its destination register depending on the value of the predicate [74]. This problem is called the *multiple definition problem* in [19]. Figure 2.4 demonstrates an example of the multiple definition problem. In this example, instructions will write a value into register R33 depending on the predicate (P6). The ADD instruction does not know which value will be in R33 until the predicate value is known.

Several solutions have been proposed to handle this problem: converting predicated instructions into C-style conditional expressions [74], breaking predicated instructions into two μops [21], the select-μop mechanism [79], and predicate prediction [19].[2]

2.2.1 Converting a Predicated Instruction Into a C-style Conditional Expression

A predicated instruction is transformed into another instruction similar to a C-style conditional expression. For example, `(P6) MOV R33 = 1` instruction is converted to

[2] In this dissertation, the C-style conditional expression mechanism is used as the baseline processor mechanism.



```
A  if (cond1) {
B    b = 1;
     if (cond2) {
D      d = 1;
     }
C  else {
     b = 0;
   }
G  c = b+d;
```

(code)

```
A          p1 = cond1
           branch p1, TARGET1
C          mov b, 0
           uncond br. TARGET2
B  TARGET1:
           mov b,1
           p2 = cond2
           branch !p2 TARGET2
D          mov d,1
G  TARGET2:
           add c, b, d
```

```
        p1 = cond1
(p1)    mov b,1
        p2 = p1 & (cond2)
(p2)    mov d, 1
(!p1)   mov b, 0
        add c, b, d
```

(a) (b)

Figure 2.3: Nested `if-else` source code and the corresponding assembly code for (a) normal branch code (b) predicated code

the μop `R33 = P6 ? 1 : R33`. If the predicate is TRUE, the instruction performs the computation and stores the result into the destination register. If the predicate is FALSE, the instruction simply moves the old value of the destination register into its destination register, which is architecturally a NOP operation. Hence, regardless of the predicate value, the instruction *always* writes into the destination register, allowing the dependent instructions

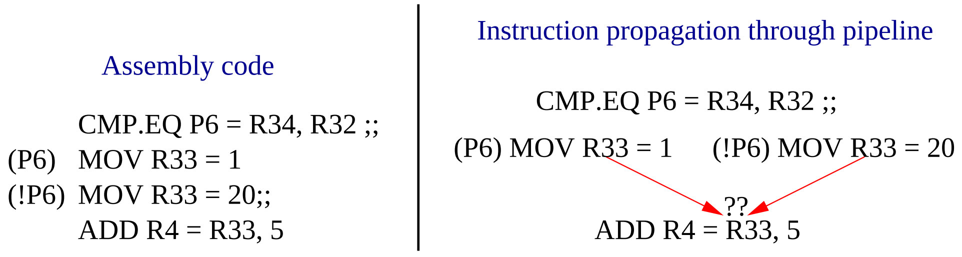


Figure 2.4: An example of the multiple definition problem [19]

to be renamed correctly. This mechanism requires four register sources (the old destination register value, the source predicate register, and the two source registers).

2.2.2 Breaking a Predicated Instruction into Two μops

The CMOV instruction in the Alpha ISA behaves like the C-style conditional expression. For example, `CMOV Ra, Rb, Rc` is the same as `Rc = Ra ? Rb : Rc`. This mechanism requires an extra input source, which results in an extra input source only for the CMOV instruction. To remove this special case, Alpha 21264 decomposes the CMOV instruction into two 2-operand instructions [21].

The Alpha architecture instruction	CMOV Ra, Rb, Rc
Becomes the 21264 instructions	CMOV1 Ra, oldRc ⇒ newRc1
	CMOV2 newRc1, Rb ⇒ newRc2

The first instruction, CMOV1, tests the value of Ra and records the result of this instruction in a 65th bit of its destination register, newRc1, which is a temporary physical register. It also copies the value of the old physical destination register, oldRc, to newRc1. The second instruction, CMOV2, then copies either the value in newRc1 or the value in Rb into a second physical destination register, newRc2, based on the CMOV predicate bit stored in the 65th bit of newRc1.

The negative effect of this mechanism is that it increases the number of μops since every CMOV instruction becomes two micro-ops.

2.2.3 The Select-μop Mechanism

To reduce the number of extra μops, Wang et al.[79] proposed the select-μop mechanism. Similar to the static single assignment (SSA) form, a select-μop is inserted to select between multiple renamed registers based on the guarding predicate value. Multiple renamed registers and their guarding predicates are assigned as the source operands of the select-μop. A new renamed register allocated for the result of the select-μop can then be referenced by all subsequent consumer instructions. The code in Figure 2.4 has two instructions which write different values in architectural register R33. For example, architectural register R33 in instruction (`(P6) MOV R33 = 1`) is allocated to physical register PR10 and architectural register R33 in instruction (`(!P6) MOV R33 = 1`) is allocated to physical register PR20. The select-μop mechanism inserts a select-μop instruction (`PR30 = P6 ?  PR10 :  PR20`) to choose between two physical registers. The select-μop will write the result into a new physical register (PR30) when the predicate value is evaluated. The select-μop mechanism also updates the register alias table, so younger instructions source PR30 for architecture register R33.

The select-μop mechanism could reduce the number of μops by combining multiple CMOV instructions to one select-μop when there are several instructions that have the same destination registers but different predicate values [79].

2.2.4 Predicate Prediction

Chuang and Calder [19] proposed a predicate predictor to solve the multiple definition problem. The predicate value is predicted at the beginning of the renaming stage so only the instructions whose predicate values are predicted to be TRUE are renamed and passed to the pipeline. If the prediction is wrong, the replay mechanism re-renames the registers and re-executes dependent instructions. In the example in Figure 2.4, when the processor fetches instruction (`(P6) MOV R33 = 1`) the processor predicts P6 value. If P6 value is predicted as TRUE, the processor sends instruction (`(P6) MOV R33 = 1`) into the pipeline but not instruction (`(!P6) MOV R33 = 20`). Instruction (`(!P6) MOV R33 = 20`) is still fetched but it will be stored in a separate buffer until the predicate value is resolved. Hence, instruction (`ADD R4 = R33, 5`) sources the result of instruction (`(P6) MOV R33 = 1`). Later P6 value is evaluated and if it turns out to be FALSE, the processor fetches instruction (`(!p6) MOV R33 = 20`) from the buffer and executes it. Instruction (`ADD R3 = R33, 5`) also will be re-executed.

2.3 The Overhead of Predicated Execution

Predicated execution introduces two major sources of overhead on the dynamic execution of a program compared to conditional branch prediction. First, the processor needs to fetch additional instructions that are guaranteed to be useless since their predicates will be FALSE. These instructions waste fetch and possibly execution bandwidth and occupy processor resources that can otherwise be utilized by useful instructions. Second, an instruction that is dependent on a predicate value cannot be executed until the predicate value it depends on is ready. This introduces additional delay into the execution of predicated instructions and their dependents, and hence may increase the execution time of the program. We analyze the performance impact of these two sources of overhead on an out-of-order processor model that implements predicated execution. The simulation methodology and the baseline machine are described in Chapter 4.

Figure 2.5 shows the performance improvement achievable if the sources of overhead in predicated execution are ideally eliminated. Data is normalized to the execution

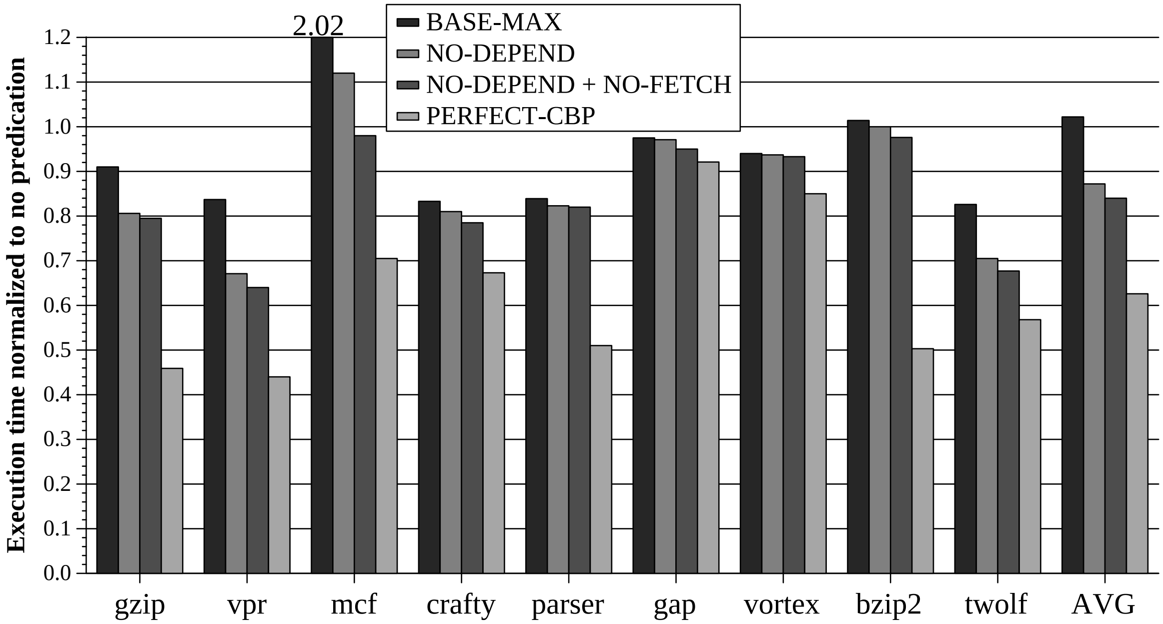


Figure 2.5: Execution time when sources of overhead in predicated execution are ideally eliminated.

time of the non-predicated code binary. For each benchmark, four bars are shown from left to right: (1) BASE-MAX shows the execution time of the predicated code binary produced by the ORC compiler [57] - with all overheads of predicated execution faithfully modeled. (2) NO-DEPEND shows the execution time of the predicated code binary when the dependencies due to predication are ideally (using oracle information) removed. (3) NO-DEPEND + NO-FETCH shows the execution time of the predicated code binary when both sources of overhead in predicated execution are ideally eliminated: in addition to predicate dependencies, the instructions whose predicates are FALSE are ideally eliminated so that they do not consume fetch and execution bandwidth. (4) PERFECT-CBP shows the execution time of the non-predicated code binary when all conditional branches are perfectly predicted using oracle information. This figure shows that predicated execution helps many benchmarks, but it does not improve the *average execution time* over a non-predicated code binary when its overhead is faithfully modeled (i.e., the average execution time of BASE-

MAX is 1.02, which is 2% longer than that of the baseline (no predication)). However, if the sources of overhead associated with it are completely eliminated, predicated execution would improve the average execution time by 16.4% over no predication. When the overhead of predicated execution is eliminated (NO-DEPEND+NO-FETCH), the predicated code binary outperforms the non-predicated code binary by more than 2% *on all benchmarks*, even on those where predicated execution normally loses performance (i.e., mcf and bzip2). Note that a significant performance difference still exists between NO-DEPEND + NO-FETCH and PERF-CBP (Perfect conditional branch prediction improves the average execution time by 37.4%). This is due to the fact that not all branches can be eliminated using predication. For example, backward (loop) branches, which constitute a significant proportion of all branches, cannot be eliminated using predicated execution [3, 16].

Chapter 3

Related Work

Many researchers have studied how to handle branch instructions. This chapter classifies the relevant approaches into three categories, predicated execution, control-flow independence, and multipath execution, and briefly describes the proposed approaches.

3.1 Related Research on Predicated Execution

Predicated execution was first implemented in the Cray-1 computer system as *mask vectors* [67]. Allen and Kennedy et al. proposed the predication of instructions using *if conversion* to enable automatic vectorization in the presence of complex control flow [3]. Hsu and Davidson proposed the use of predicated execution for scalar instructions, which they called *guarded execution*, to reduce the penalty of conditional branches in deeply-pipelined processors [33]. Hsu and Davidson also described how predicated execution enables compiler-based code scheduling optimizations.

Several papers examined the impact of predicated execution on branch prediction and instruction-level parallelism. Pnevmatikatos and Sohi [61] showed that predicated execution can significantly increase a processor's ability to extract parallelism, but they also showed that predication results in the fetch and decode of a significant number of useless instructions. Mahlke et al. [50], Tyson [78], and Chang et al. [10] showed that predicated execution can eliminate a significant number of branch mispredictions and can therefore reduce the program execution time.

Choi et al. [16] examined the performance advantages and disadvantages of predicated execution on a real IA-64 implementation. They showed that even though predication can potentially remove 29% of the branch mispredictions in the SPEC CPU2000 INT benchmark suite, it results in only a 2% improvement in average execution time. For some benchmarks, a significant performance loss is observed with predicated execution. The performance loss in some benchmarks and the small performance gain in others are due to the overhead of predicated execution.

3.1.1 Overcoming the Problems of Predicated Execution

3.1.1.1 Control-Flow Limitation Problem

Hyperblock formation [51] predicates frequently executed basic blocks based on profiling data. 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 [50]. However, 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 [70].

3.1.1.2 The Lack of Adaptivity Problem

Hazelwood and Conte [31] discussed the performance problems associated with predicated code when the input set of the program changes. They used dynamic profiling to identify hard-to-predict branches at run-time to solve this problem. Their mechanism dynamically converted the identified hard-to-predict branches to predicated code via a dynamic optimization framework. They sampled the individual branch misprediction rates at the beginning of the program to identify most of the hard-to-predict branches for a given input set. Besides the overhead of dynamic profiling and dynamic optimization, with their mechanism, they can decide when to use predicated code only based on a given input set. In contrast to their mechanism, both wish branches and DMP can decide when to use predicated code based on a control-flow path which leads to a branch or program phase not only for a given input set.

3.1.1.3 Predicate Prediction

Chuang and Calder [19] proposed a hardware mechanism to predict *all* predicate values in order to overcome the register renaming problem in an out-of-order processor that implements predicated execution. Although they did not mention it, their mechanism can also reduce the extra instruction overhead of predicated execution. With predicate prediction, instructions whose predicates are predicted FALSE do not need to be executed, thus reducing the overhead of predicated execution—provided the prediction is correct. However, the processor still needs to fetch and decode all the predicated instructions. The adaptive predicated execution paradigm can eliminate the fetch and decode of predicated instructions, as well as their execution. Also, every predicate is predicted with predicate prediction, which can result in performance loss for hard-to-predict predicates. Furthermore, both wish branches and DMP can eliminate the misprediction penalty for backward (loop) branches, whereas conventional predication augmented with predicate prediction cannot. Recently, Quinones et al. [62] proposed a selective predicate prediction mechanism. With the selective predicate predictor, the processor predicts a predicate value only if the predicate prediction has high confidence. Hence, the processor can reduce the execution bandwidth of predicated-FALSE instructions if the predicate prediction is correct. However, although their mechanism can overcome the problem of predicting every predicate value, it does nothing for the rest of the problems stated above.

3.1.1.4 Lack of ISA support

Klauser et al. [43] proposed *dynamic hammock predication*, which is a purely hardware mechanism that dynamically predicates hammock branches. Like wish branches and dynamic predication, dynamic hammock predication also enables the hardware to dynamically decide whether or not to use predication for a hammock branch. In contrast to wish branches and DMP, dynamic hammock predication is a purely hardware-based mechanism. In the wish branch mechanism, the compiler generates predicated code. In the dynamic predication mechanism, the compiler provides the hints about control flow information. Both wish branches and DMP do not require complex hardware to construct control flow information. Furthermore, dynamic hammock predication allows only simple control-flow graphs to be converted into predicated code whereas both the wish branch mechanism and the dynamic predication mechanism can predicate a *wider* range of control flow shapes.

Santos et al. [26, 25] proposed *Dynamic Conditional Execution (DCE)*, a hybrid mechanism of dynamic predication and multipath execution to handle complex branch instructions. In DCE, instructions that are not on correct paths become NOPs just like in dynamic hammock predication. However, after the processor joins at the corresponding control-flow merge point, the processor generates replicated instructions to solve data-flow dependences. Hence, the overhead of DCE is very similar to multipath[1] execution except that in DCE only instructions data-dependent on the instructions inside a hammock are replicated (and thus executed) multiple times. In contrast, in multipath execution, all the instructions are fetched/executed multiple times. Furthermore, DCE can handle only simple and nested hammocks, whereas DMP can handle more complex control flow graphs.

[1] Multipath execution is described in Section 3.3.

3.1.2 Predicated Code Generation Algorithms

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 [61] and Tyson [78] 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 [10].

Hyperblock formation [51] uses path execution frequencies, basic block sizes, and basic block characteristics to decide which blocks should be included in a hyperblock. With hyperblocks, one of the major benefits of predicated code is due to increased basic block sizes, which enhances the compiler's scope for code optimization. Hence, identifying hot-paths is more important than identifying highly mispredicted branches. August et al. [6] 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 [81].

Mantripragada and Nicolau [53] 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.

Unlike static predication, the adaptive predication paradigm does not require comprehensive compiler algorithms since a bad compiler's decision can be corrected later by hardware at run-time. Nonetheless, since the compiler has more information about control flows, a simple cost-benefit analysis or heuristics to generate predicated code or mark diverge branches still could help improve the performance of the program. We will show the compiler algorithms and heuristics we use to generate wish branch code in Section 4.4.2. We will present the compiler algorithms and heuristics to mark diverge branches/CFM points and a new analytical model to select candidates for frequently-hammocks and loops which cannot be predicated by conventional if-conversion in Chapter 6.

3.2 Related Work on Control Flow Independence

Several hardware mechanisms have been proposed to exploit control flow independence [65] by reducing the branch misprediction penalty or improving parallelism [65, 18, 15]. These techniques aim to avoid flushing the processor pipeline when the processor is known to be at a control-independent point in the program at the time a branch misprediction is signaled. In contrast to both wish branches and DMP, these mechanisms require a significant amount of hardware to exploit control flow independence [65]. Hardware is required for the following:

1. *Detection of the reconvergent (control-flow independent) point in the instruction stream*: While some mechanisms use software to detect the reconvergent point [65, 66], most proposed mechanisms use hardware-based heuristics and predictors [64, 18, 15, 28, 20]. The hardware used to detect/predict the reconvergent point adds more complexity to the processor pipeline. In contrast, a wish branch exactly specifies the reconvergent point, because the compiler that generates the wish branch knows *exactly* where the reconvergent point is in the instruction stream. In DMP, the compiler specifies reconvergent points using special instructions. Hence, there is no need for extra hardware.
2. *Removal of wrong-path instructions, formation of correct data dependences for control-independent instructions, and selective re-scheduling and re-execution of instructions*: Proposed mechanisms to exploit control flow independence [64, 65, 66, 18, 15, 28] require fairly complicated hardware structures to accomplish these tasks. In contrast, as both wish branches and DMP make use of predication to exploit control-

flow independence, there is no need to provide extra hardware other than what is in place to support predicated execution. Instructions that are on the wrong-path will become NOPs because they are predicated, and the control-independent instructions on the correct path already have the correct data dependences because the compiler correctly identifies their dependences while generating predicated code, which eliminates the need for re-scheduling and re-execution. In summary, both wish branches and DMP, with their use of predication, eliminate most of the complex hardware support required to exploit control-flow independence purely in hardware.

3. *Formation of correct data dependences for control independent instructions*: Although instructions may be control independent with a preceding block of instructions, they may not be *data independent*. Therefore, a hardware mechanism that selectively flushes one of the previous blocks needs to fix the data dependencies (both register and memory) for the instructions in the later control-independent blocks. Previously-proposed mechanisms have all devoted a significant amount of hardware to accomplish this [64, 65, 66, 18, 15, 28]. As wish branches and DMP make use of predication to exploit control-flow independence, there is no need to fix the data dependences for control independent instructions. These instructions already have the *correct* data dependences, because the compiler correctly identifies their source instructions while generating predicated code.

4. *Selective re-scheduling and re-execution of instructions*: A control independent instruction that got the wrong source data value due to a false data dependence with a wrong-path instruction needs to be re-scheduled and re-executed in previously proposed hardware mechanisms [64, 65, 66, 18, 15, 28]. Such selective re-scheduling also requires hardware and adds complexity to the instruction scheduling logic. In the adaptive predicated execution, the need for re-scheduling is eliminated. Since the instructions that could possibly be on the wrong-path are predicated by the compiler, the control-independent instructions can never get the wrong source data value and,

therefore, never need to be re-scheduled.

3.3 Related Work on Multipath Execution

Starting with Riseman and Foster's eager execution [63] and the dual-path fetch in the IBM 360/91 [4], several contributions have been made in the field of multipath execution. Uht's survey of multipath execution [47] provides a good overview and comparisons. This section will only review the work most relevant to wish branches and DMP.

Heil and Smith [32] and Farrens et al. [27] proposed selective/limited dual path execution mechanisms. The processor starts fetching from both paths of a low confidence branch. The following low confidence branch either delays dual-path execution or is ignored until the first low confidence branch is resolved. When the low confidence branch is resolved, the instructions on the mispredicted path are discarded. As we will show in Section 5.5, dual-path execution's performance improvement is not as significant as that of DMP or wish branches 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. [45] as an implementation of multipath execution. Multipath execution requires more hardware 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 will show in Section 5.5.5, multipath execution significantly increases maximum power and energy consumption without providing as large performance improvements as those of DMP.

Chapter 4

Wish Branches

This chapter presents a set of new control flow instructions, called *wish branches*, the first mechanism of adaptive predicated execution. With wish branches, we can combine normal conditional branching with predicated execution, providing the benefits of predicated execution without its wasted fetch and execution bandwidth. Wish branches aim to reduce the branch misprediction penalty by using predicated execution only when it increases performance. The decision of when to use predicated execution is made during run-time using a branch predictor and a confidence estimator. While in some run-time scenarios normal branches perform better than predicated execution, predicated execution performs better in others. Wish branches aim to get the better of the two under all scenarios.

A wish branch looks like a normal branch but the code on the fall-through path between the branch and the target is predicated. A forward wish branch is called a *wish jump*. When the processor fetches the wish jump, it predicts the direction of the wish jump using a branch predictor, just like it does for a normal branch. If the wish jump is predicted not-taken, the processor executes the predicated code. But if it is mispredicted, the pipeline does not need to be flushed since the fall-through path is predicated. If the wish jump is predicted taken, the processor executes the normal branch code. If this prediction is correct, the extra useless instructions in the predicated code are not fetched. Hence, a wish jump can obtain the better performance of a normal branch and predicated execution. Wish jumps are used with a confidence estimator. When the confidence estimator predicts that a wish jump might be mispredicted, the hardware performs predicated execution. Thus, the wish jump mechanism gives the hardware the option to dynamically decide whether or not to use predicated execution.

A backward loop branch can be converted to a wish branch instruction, which we call a *wish loop*. The wish loop instruction can reduce the branch misprediction penalty by exploiting the benefits of predicated execution for backward branches. To use the wish loop, the compiler predicates the body of the loop using the loop branch condition as the predicate. When the wish loop is mispredicted, the processor doesn't need to flush the pipeline because the body of the loop is predicated.

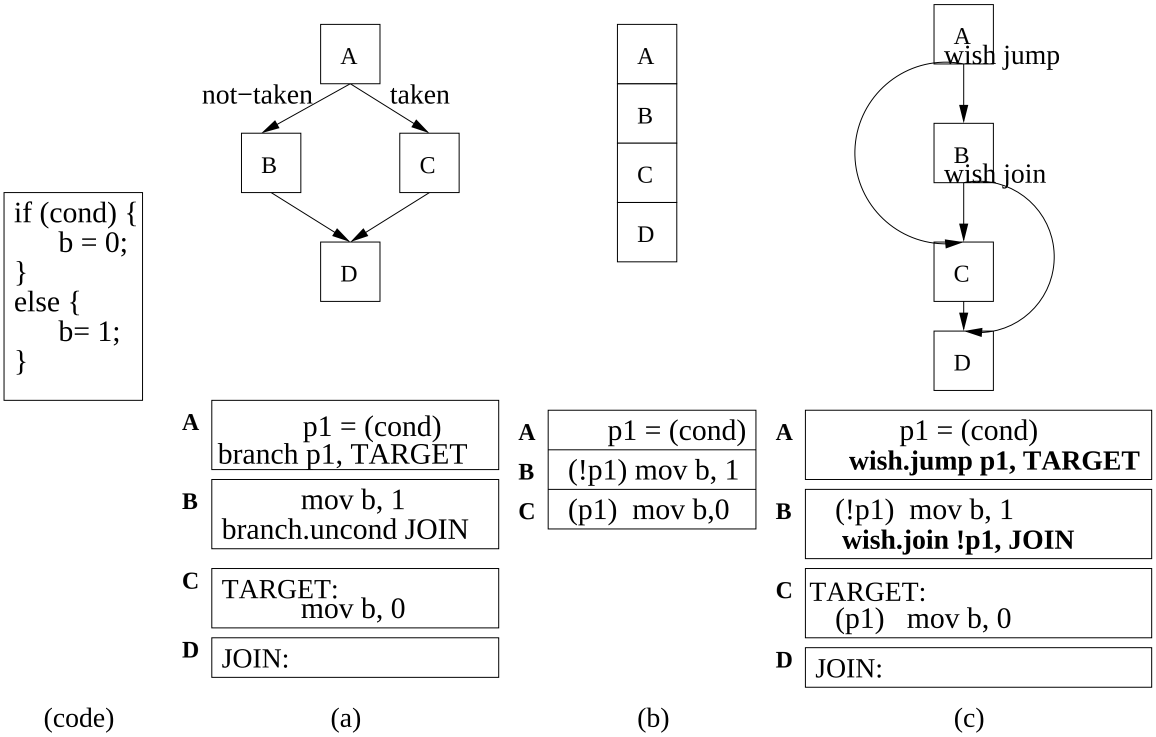
This chapter describes the semantics, types, and operation of wish branches and evaluates the performance of wish branches in the IA-64 ISA.

4.1 Wish Branches

There are three different wish branch instructions: (1) wish jumps (Section 4.1.1), (2) wish joins (Section 4.1.1), and (3) wish loops (Section 4.1.2). Wish jumps and wish joins are for forward branches and wish loops are used for backward branches. We will explain the behavior of wish branches and how wish branches are different from normal branches and predicated execution in this section.

4.1.1 Wish Jumps and Wish Joins

Figure 4.1 shows a simple source code example and the corresponding control flow graphs and assembly code for: (a) a normal branch, (b) predicated execution, and (c) a wish jump/join. The main difference between the wish jump/join code and the normal branch code is that the instructions in basic blocks B and C are predicated in the wish jump/join code (Figure 4.1c), but they are not predicated in the normal branch code (Figure 4.1a). The first conditional branch in the normal branch code is converted to a wish jump instruction and the following control-dependent unconditional branch is converted to a wish join instruction in the wish jump/join code. The difference between the wish jump/join code and the predicated code (Figure 4.1b) is that the wish jump/join code has branches (i.e., the wish jump and the wish join), but the predicated code does not.



```
if (cond) {
     b = 0;
}
else {
     b= 1;
}
```

(a)

```
A   p1 = (cond)
    branch p1, TARGET
B   mov b, 1
    branch.uncond JOIN
C   TARGET:
        mov b, 0
D   JOIN:
```

(b)

```
A        p1 = (cond)
B   (!p1) mov b, 1
C   (p1)  mov b,0
```

(c)

```
A        p1 = (cond)
     wish.jump p1, TARGET
B   (!p1)  mov b, 1
     wish.join !p1, JOIN
C   TARGET:
    (p1)   mov b, 0
D   JOIN:
```

Figure 4.1: Source code and the corresponding control flow graphs and assembly code for (a) normal branch code (b) predicated code (c) wish jump/join code.

Wish jump/join code can be executed in two different modes (*high-confidence-mode* and *low-confidence-mode*) at run-time. The mode is determined by the confidence of the wish jump prediction. When the processor fetches the wish jump instruction, it generates a prediction for the direction of the wish jump using a branch predictor, just like it does for a normal conditional branch. A hardware confidence estimator provides a confidence estimation for this prediction. If the prediction has high confidence, the processor enters high-confidence-mode for this branch. If it has low confidence, the processor enters low-confidence-mode.

High-confidence-mode is the same as using normal conditional branch prediction. To achieve this, the wish jump instruction is predicted using the branch predictor. The source predicate value (p1 in Figure 4.1c) of the wish jump instruction is predicted based on the predicted branch direction so that the instructions in basic block B or C can be executed before the predicate value is ready. When the wish jump is predicted to be taken, the predicate value is predicted to be TRUE (and block B, which contains the wish join, is not fetched). When the wish jump is predicted to be not taken, the predicate value is predicted to be FALSE and the wish join is predicted to be taken.

Low-confidence-mode is the same as using predicated execution, except it has additional wish branch instructions. In this mode, the wish jump and the following wish join are always predicted to be not taken. The source predicate value of the wish jump instruction is not predicted and the instructions that are dependent on the predicate only execute when the predicate value is ready.

When the confidence estimation for the wish jump is accurate, either the overhead of predicated execution is avoided (high confidence) or a branch misprediction is eliminated (low confidence). When the wish jump is mispredicted in high-confidence-mode, the processor needs to flush the pipeline just like in the case of a normal branch misprediction. However, in low-confidence-mode, the processor never needs to flush the pipeline, even when the branch prediction is incorrect. Like predicated code, the instructions that are not on the correct control flow path will become NOPs since all instructions that are control-dependent on the branch are predicated.

4.1.2 Wish Loops

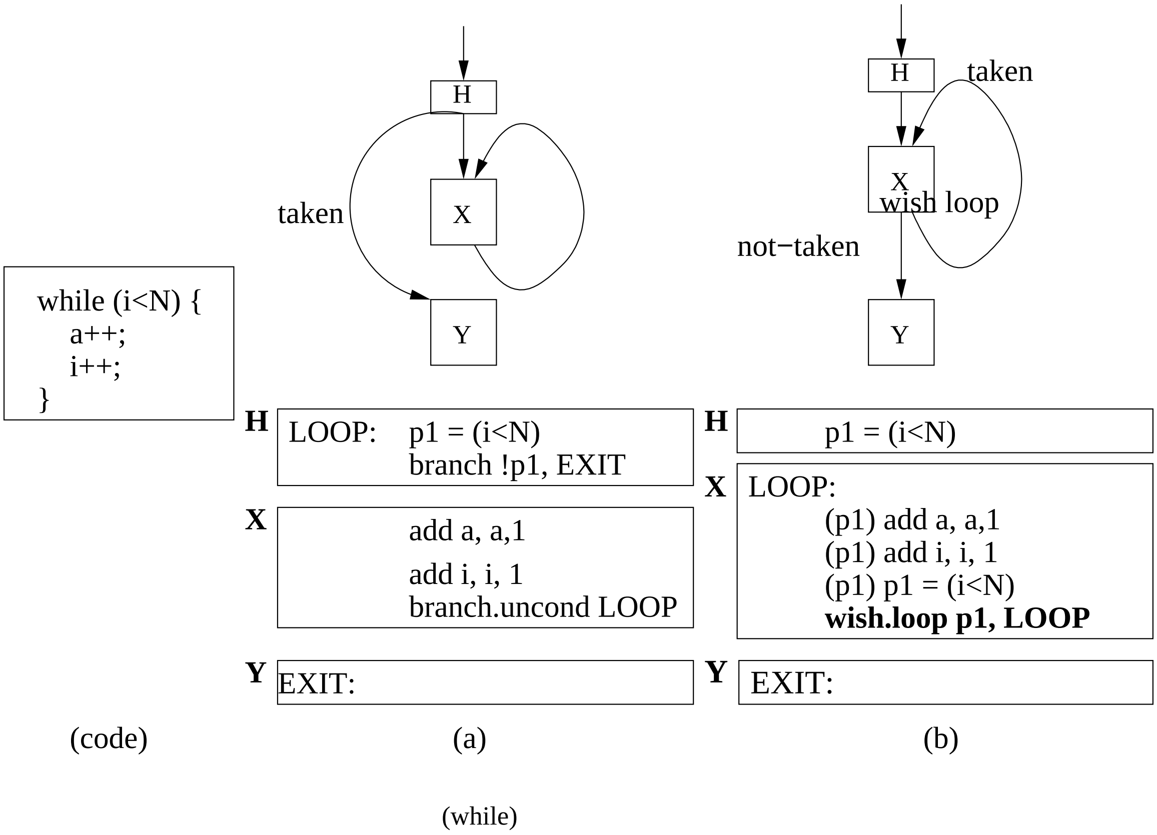
A wish branch can also be used for a backward branch. We call this a *wish loop* instruction. Figure 4.2 contains the source code for a simple loop body and the corresponding control-flow graphs and assembly code for: (a) a normal backward branch and (b) a wish loop. We compare wish loops only with normal branches since backward branches cannot be directly eliminated using predication [3]. A wish loop uses predication to reduce the branch misprediction penalty of a backward branch without eliminating the branch.

The main difference between the normal branch code (Figure 4.2a) and the wish loop code (Figure 4.2b) is that in the wish loop code the instructions in block X (i.e., the loop body) are predicated with the loop branch condition. Wish loop code also contains an extra instruction in the loop header to initialize the predicate to 1 (TRUE). To simplify the explanation of the wish loops, we use a `do-while` loop example in Figure 4.2. Similarly, a `while` loop as shown in Figure 4.3 or a `for` loop can also utilize a wish loop instruction.

When the wish loop instruction is first encountered, the processor enters either high-confidence-mode or low-confidence-mode, depending on the confidence of the wish loop prediction.

In high-confidence-mode, the processor predicts the direction of the wish loop according to the loop/branch predictor. If the wish loop is predicted to be taken, the predicate value (p1 in Figure 4.2b) is predicted to be TRUE, so the instructions in the loop body can be executed without waiting for the predicate to be evaluated. If the wish loop is mispredicted in high-confidence-mode, the processor flushes the pipeline, just as in the case of a normal branch misprediction.

If the processor enters low-confidence-mode, it stays in this mode until the loop is exited. In low-confidence-mode, the processor still predicts the wish loop according to the loop/branch predictor. However, it does *not* predict the predicate value. Hence, the



```
while (i<N) {
    a++;
    i++;
}
```

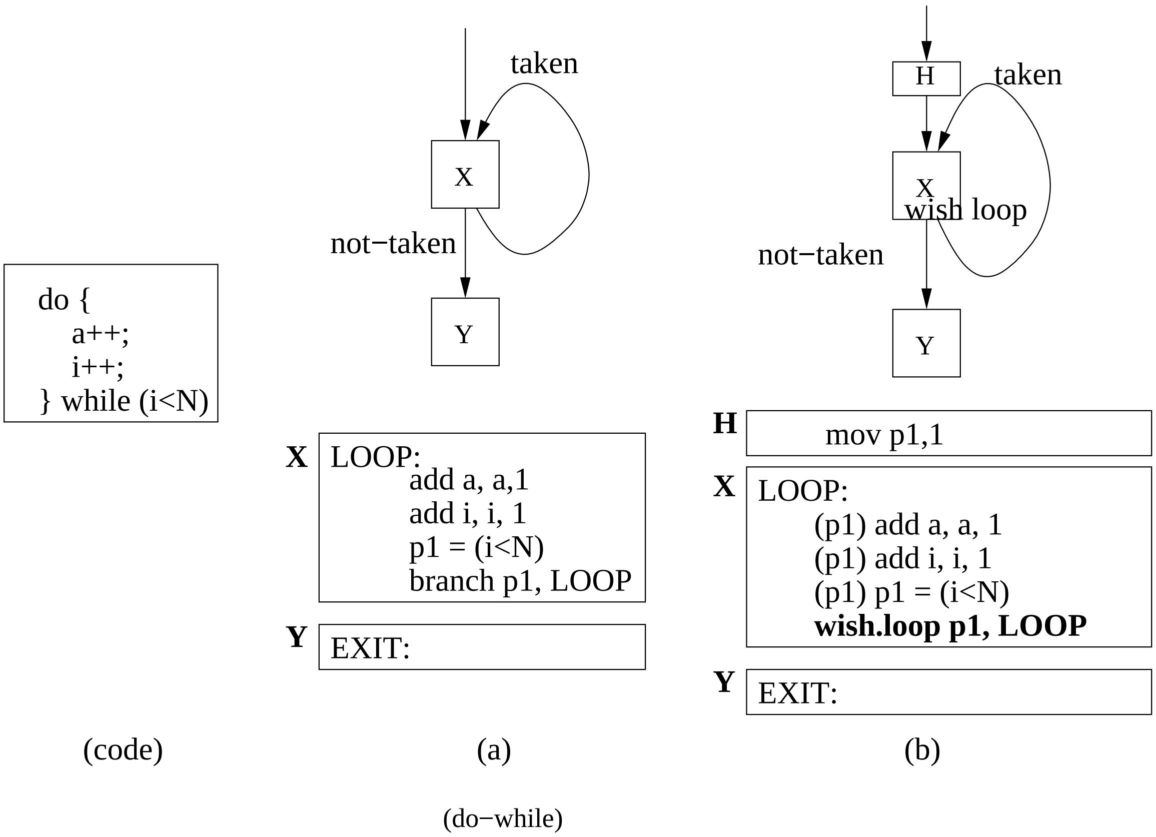
```
LOOP:   p1 = (i<N)
        branch !p1, EXIT

        add a, a,1
        add i, i, 1
        branch.uncond LOOP

EXIT:
```

```
        p1 = (i<N)

LOOP:
        (p1) add a, a,1
        (p1) add i, i, 1
        (p1) p1 = (i<N)
        wish.loop p1, LOOP

EXIT:
```

Figure 4.2: `while` loop source code and the corresponding control flow graphs and assembly code for (a) normal backward branch code (b) wish loop code.

iterations of the loop are predicated (i.e., fetched but not executed until the predicate value is known) during low-confidence-mode. There are three misprediction cases in this mode: (1) *early-exit*: the loop is iterated fewer times than it should be, (2) *late-exit*: the loop is iterated only a few more times by the processor front end than it should be and the front end has already exited when the wish loop misprediction is signaled, and (3) *no-exit*: the loop is still being iterated by the processor front end when the wish loop misprediction is signaled (as in the late-exit case, it is iterated more times than needed).

For example, consider a loop that iterates 3 times. The correct loop branch direc-



```
do {
    a++;
    i++;
} while (i<N)
```

```
X  LOOP:
          add a, a,1
          add i, i, 1
          p1 = (i<N)
          branch p1, LOOP

Y  EXIT:
```

```
H        mov p1,1

X  LOOP:
         (p1) add a, a, 1
         (p1) add i, i, 1
         (p1) p1 = (i<N)
         wish.loop p1, LOOP

Y  EXIT:
```

(code)

(a)

(b)

(do−while)

Figure 4.3: `do-while` loop source code and the corresponding control flow graphs and assembly code for (a) normal backward branch code (b) wish loop code.

tions are TTN (taken, taken, not-taken) for the three iterations, and the front end needs to fetch blocks $X_1X_2X_3Y$, where X_i is the i^{th} iteration of the loop body. An example for each of the three misprediction cases is as follows: In the early-exit case, the predictions for the loop branch are TN, so the processor front end fetches blocks X_1X_2Y. One example of the late-exit case is when the predictions for the loop branch are TTTTN so the front end fetches blocks $X_1X_2X_3X_4X_5Y$. For the no-exit case, the predictions for the loop branch are TTTTT...T so the front end fetches blocks $X_1X_2X_3X_4X_5...X_N$.

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

In the late-exit case, the fall-through block Y has been fetched before the predicate for the first extra block X_4 has been resolved. Therefore, it is more efficient to simply allow X_4 and subsequent extra block X_5 to flow through the data path as NOPs (with predicate value p1 = FALSE) than to flush the pipeline. In this case, the wish loop performs better than a normal backward branch because it reduces the branch misprediction penalty. The smaller the number of extra loop iterations fetched, the larger the reduction in the branch misprediction penalty.

In the no-exit case, the front end has not fetched block Y at the time the predicate for the first extra block X_4 has been resolved. Therefore, it makes more sense to flush X_4 and any subsequent fetched extra blocks, and then fetch block Y, similar to the action taken for a normal mispredicted branch. We could let $X_4X_5...X_N$ become NOPs as in the late-exit case, but that would increase energy consumption without improving performance.

4.1.2.1 More on Wish Loops and Predication

Traditional predicated code reduces the branch misprediction penalty by eliminating branches. Since backward (loop) branches cannot be eliminated with predication due to the nature of the control flow [3], traditional predicated execution cannot eliminate or reduce the branch misprediction penalty for backward branches. However, with wish branches, in the presence of the branch (which is the wish branch itself), the processor can still reduce the branch misprediction penalty using predicated code as we showed in the late-exit case for the wish loop. Wish branches reduce the branch misprediction penalty not by eliminating branches but by using the characteristics of predicated code: instructions that should not be executed will become NOPs when the predicate value becomes available (as false). Hence, with wish loops and predicated code, wish branches can reduce the branch misprediction penalty due to backward (loop) branches without having to eliminate such branches.

Loop unrolling can reduce the number of loop branches, thereby perhaps reducing the number of loop branch mispredictions. With predication, the compiler could perform loop unrolling more aggressively. However, loop unrolling still cannot eliminate all backward branches. The remaining backward branches can still be mispredicted. Wish branches can therefore convert the remaining branches into wish loops. Furthermore, loop unrolling increases the pressure on architectural registers, increases code size, and requires extra code to handle loop iterations that are not a multiple of the unrolling factor - three sources of complexity that do not exist with wish branches. In addition, loop unrolling is usually useful for regular loops that iterate a large number of times, whereas wish loop instructions are aimed at eliminating loop branch mispredictions in loops that iterate a small number of times - loops that occur frequently in irregular integer programs.

Note that the compiler can also predicate instructions inside the loop body to facilitate Software Pipelining (SWP). SWP can be used without predication also. However, Warter et al. [80] showed that pipelined loops performed 34% faster on average with predication than without predication. The purpose of software pipelining is orthogonal to the purpose of wish branches: SWP is used to increase instruction level parallelism in order to make static scheduling more effective (by finding more independent instructions across different loop iterations that can be scheduled in parallel) whereas wish branches are used to reduce the branch misprediction penalty. As such, software pipelining is much less beneficial on processors that support dynamic scheduling whereas wish branches still provide significant performance improvements on dynamically-scheduled processors. Note that both loop unrolling and software pipelining, as shown by Choi et al. [16], are less effective for irregular integer benchmarks where parallelism is hard to find at compile time, In contrast, wish branches are more effective for such irregular benchmarks where the branch prediction accuracy is relatively low.

For these reasons, we claim that wish loops can reduce the branch misprediction penalty for backward (loop) branches, which cannot be reduced by traditional predicated execution. While traditional predication facilitates better loop unrolling and software pipelining, these mechanisms are not fundamentally aimed at reducing the branch misprediction penalty (even though loop unrolling sometimes can, as a side effect). Hence, wish branches are orthogonal to these two schemes and can be combined with them to provide higher performance.

4.1.3 Wish Branches in Complex Control Flow

Wish branches are used not only for simple control flow. They can also be used in complex control flow where there are multiple branches, some of which are control-dependent on others. Figure 4.4 shows a code with complex control flow, and the control flow graphs of the normal branch code, predicated code, and the wish branch code corresponding to it.

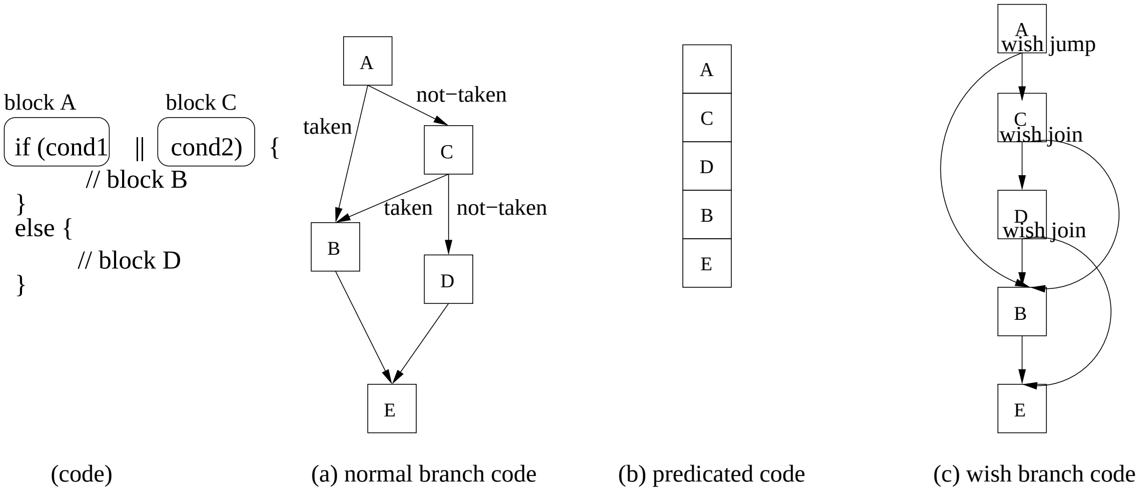


Figure 4.4: Control flow graph examples with wish branches.

When there are multiple wish branches in a given region, the first wish branch is a wish jump and the following wish branches are wish joins. Wish join instructions are control-flow dependent on earlier wish branch instructions. Hence, the prediction for a wish join is dependent on the confidence estimations made for the previous wish jump, any previous wish joins, and the current wish join itself. If the previous wish jump, any of the previous wish joins, or the current wish join is low-confidence, the current wish join is predicted to be not-taken. Otherwise, the current wish join is predicted using the branch predictor. An example of the predictions made for each of the wish branches in Figure 4.4c is shown in Table 4.1.

Table 4.1: The prediction of multiple wish branches in Figure 4.4c.

confidence			prediction		
jump (A)	join (C)	join (D)	jump (A)	join (C)	join (D)
high	high	high	predictor	predictor	predictor
high	high	low	predictor	predictor	not-taken
high	low	-	predictor	not-taken	not-taken
low	-	-	not-taken	not-taken	not-taken

4.2 Support for Wish Branches

4.2.1 ISA Support

We assume that the baseline ISA to which wish branches are to be added supports predicated execution. If the current ISA already has unused hint bits for the conditional branch instruction, like the IA-64 [34], wish branches can be implemented using the hint bit fields without modifying the ISA. Figure 4.5 shows a possible instruction format for the wish branch. A wish branch can use the same opcode as a normal conditional branch, but its encoding has two additional fields: *btype* and *wtype*. If the processor does not implement the hardware support required for wish branches, it can simply treat a wish branch as a normal branch (i.e., ignore the hint bits). New binaries containing wish branches will run correctly on existing processors without wish branch support.

OPCODE	btype	wtype	target offset	p

btype: branch type (0:normal branch 1:wish branch)
wtype: wish branch type (0:jump 1:loop 2:join)
p: predicate register identifier

Figure 4.5: A possible instruction format for the wish branch.

4.2.2 Compiler Support

4.2.2.1 Compiler Support for Wish Branch Generation

A wish branch binary is an object file consisting of a mixture of wish branches, traditional predicated code, and normal branches. The compiler decides which branches are predicated, which are converted to wish branches, and which stay as normal branches based on estimated branch misprediction rates and compile-time heuristics. The compile-time decisions need to take into account the following:

1. The size and the execution time of the basic blocks that are considered for predication/wish branch code.
2. Input data set dependence/independence of the branch.
3. The estimated branch misprediction penalty.
4. The extra instruction overhead associated with predicated execution or wish branches.

For example, it may be better to convert a short forward branch which has only one or two control-dependent instructions into predicated code rather than wish branch code because wish branch code has the overhead of at least one extra instruction (i.e., the wish jump instruction). If the misprediction rate of a branch is strongly dependent on the input data set, the compiler is more apt to convert the code into wish branch code. Otherwise, the compiler is more apt to use a normal branch or convert the code into predicated code. The compiler can determine whether or not the misprediction rate is dependent on the input data with heuristics. The compiler heuristics used to decide which branches should be converted into wish branches is an important research area that we intend to investigate in future work. The heuristics are described in Section 4.4.2.

Note that wish branches provide the compiler with more flexibility in generating predicated code. With wish branches, if the compiler makes a “bad decision” at compile time, the hardware has the ability to “correct” that decision at run time. Hence, the compiler can generate predicated code more aggressively and the heuristics used to generate predicated code can be less complicated.

4.2.3 Hardware Support

Aside from the hardware to support predicated execution, wish branches require the hardware support described below.

4.2.3.1 Instruction Fetch and Decode Hardware

Instruction decode logic must be modified so that wish branch instructions can be decoded. A branch target buffer (BTB) entry is extended to indicate whether or not the branch is a wish branch and the type of the wish branch. The fetch logic requires one additional mux to override the result of the branch predictor for a wish jump or a wish join in low-confidence-mode (since a wish jump or join is always predicted not-taken in low-confidence-mode regardless of the branch predictor outcome).

4.2.3.2 Wish Branch State Machine Hardware

Figure 4.6 shows the front-end state machine that manages the various modes of a processor implementing wish branches. There are three modes: normal-mode `(00)`, low-confidence-mode `(10)`, and high-confidence-mode `(01)`. The state diagram summarizes the mode transitions that occur in the front-end of a processor supporting wish branches, based on the information provided in Sections 4.1.1 and 4.1.2. In the state diagram, "target fetched" means that the target of the wish jump/join that caused entry into low-confidence-mode is fetched.

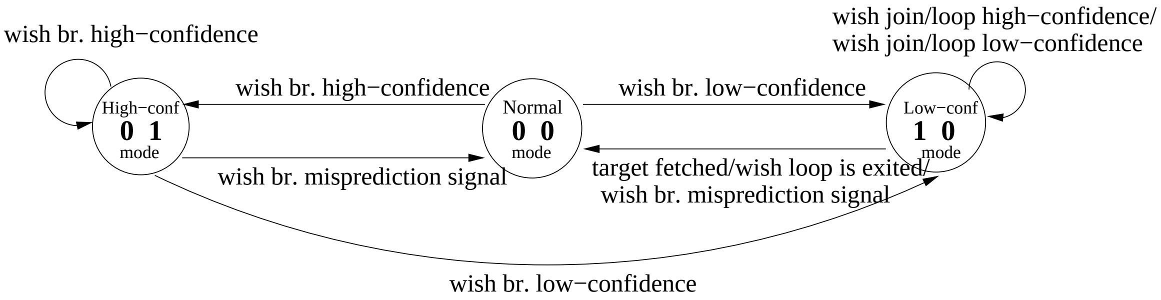

wish br. high−confidence
wish join/loop high−confidence/
wish join/loop low−confidence
wish br. high−confidence
wish br. low−confidence
High−conf
0 1
mode
Normal
0 0
mode
Low−conf
1 0
mode
wish br. misprediction signal
target fetched/wish loop is exited/
wish br. misprediction signal
wish br. low−confidence

Figure 4.6: State diagram showing mode transitions in a processor that supports wish branches.

4.2.3.3 Predicate Dependency Elimination Module

As we described in Sections 4.1.1 and 4.1.2, the predicate of the wish branch is predicted during high-confidence-mode to eliminate the delay in the execution of predicated instructions. To support this, when the processor enters high-confidence-mode, the predicate register number of the wish branch instruction is stored in a special buffer. Each following instruction compares its source predicate register number with the register number in the special buffer. If both predicate register numbers are the same, the source predicate register of the instruction is assumed to be ready, with a TRUE value when the wish branch is predicted to be taken and with a FALSE value when the wish branch is predicted to be not taken. The special buffer is reset if there is a branch misprediction or if an instruction that writes to the same predicate register is decoded.

4.2.3.4 Branch Misprediction Detection/Recovery Module

When a wish branch misprediction is detected, the processor needs to decide whether or not a pipeline flush is necessary. If the wish branch is mispredicted during high-confidence-mode,[1] the processor always flushes the pipeline. If the wish branch is mispredicted during low-confidence-mode and the wish branch is a wish jump or a wish join, then the processor does not flush the pipeline.

If a wish loop is mispredicted during low-confidence-mode, the processor needs to distinguish between early-exit, late-exit, and no-exit. To support this, the processor uses a small buffer[2] in the front end that stores the last prediction made for each static wish loop instruction that is fetched but not yet retired. When a wish loop is predicted, the predicted direction is stored in the entry corresponding to the static wish loop instruction. When a wish loop is found to be mispredicted and the actual direction is taken, then it is an early-exit case, so the processor flushes the pipeline. When a wish loop is mispredicted and the actual direction is not-taken, the branch misprediction recovery module checks the latest prediction made for the same static wish loop instruction by reading the buffer in the front end. If the last stored prediction is not taken, it is a late-exit case, because the front end must have already exited the loop, so no pipeline flush is required. If the last stored

[1] The mode that is checked when a wish branch is mispredicted is the mode of the front-end when that branch was fetched, *not* the mode of the front-end at the time the misprediction is detected.

[2] In our evaluation, we use a 4-entry fully associative buffer

prediction is taken, it is a no-exit case because the front-end must still be fetching the loop body, and the processor flushes the pipeline.[3] To reduce the hardware complexity we do not support nested wish loops.

4.2.3.5 Confidence Estimator

An accurate confidence estimator is essential to maximize the benefits of wish branches. An inaccurate confidence estimation for a wish branch can be harmful in two different ways. First, if the wish branch prediction is estimated to be low confidence even though the prediction is correct, the processor suffers from the overhead of predicated execution without any performance benefit. Second, if the wish branch prediction is estimated to be high confidence when the branch is actually mispredicted, the processor loses the opportunity to eliminate a pipeline flush.

Previously proposed confidence estimators, such as the JRS confidence estimator [35], can be used to estimate the confidence of wish branch predictions. In our evaluations, we used a tagged enhanced JRS confidence estimator [30]. Since the confidence estimator is dedicated to wish branches, its size is small. If the baseline processor already employs a confidence estimator for normal conditional branches, this estimator can also be utilized to estimate the confidence of wish branch predictions.

4.3 Advantages and Disadvantages of Wish Branches

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

[3] If the processor exited the loop and then re-entered it, this case will be incorrectly identified as a no-exit case, when it is actually a late-exit case. Hence, the processor unnecessarily flushes the pipeline, but it still functions correctly. We did not see this case happen in the benchmarks we simulated.

1. *Wish jumps/joins provide a mechanism to dynamically eliminate the performance and power overhead of predicated execution.* These instructions allow the hardware to dynamically choose between using predicated execution versus conditional branch prediction *for each dynamic instance* of a branch based on the run-time confidence estimation of the branch's prediction.
2. *Wish jumps/joins allow the compiler to generate predicated code more aggressively and using simpler heuristics, since the "bad compile-time decisions" can be corrected at run-time.* In previous research, a static branch instruction either remained as a conditional branch or was predicated for *all its dynamic instances*, based on less accurate compile-time information. If the compiler made a bad decision to predicate, there was no way to dynamically eliminate the overhead of the bad compile-time decision. For this reason, compilers have been conservative in producing predicated code and have avoided large predicated code blocks.
3. *Wish loops provide a mechanism to exploit predicated execution to reduce the branch misprediction penalty for backward (loop) branches.* In previous research, it was not possible to reduce the branch misprediction penalty for a backward branch by solely utilizing predicated execution [3, 16]. Hence, predicated execution was not applicable for a significant fraction of hard-to-predict branches.
4. *Wish branches will also reduce the need to re-compile the predicated binaries whenever the machine configuration and branch prediction mechanisms change from one processor generation to another (or even during compiler development).* A branch that is hard-to-predict in an older processor may become easy-to-predict in a newer processor with a better branch predictor. If that branch is conventionally predicated by the old compiler, the performance of the old code will degrade on the new processor because predicated execution would not improve, and in fact degrade, the performance of the now easy-to-predict branch. Hence, to utilize the benefits of the new processor, the old code needs to be recompiled. In contrast, if the branch were con-

verted to a wish branch by the compiler, the performance of the old binary would not degrade on the new processor, since the new processor can dynamically decide not to use predicated execution for the easy-to-predict wish branch. Thus, wish branches reduce the need to frequently re-compile by providing flexibility (dynamic adaptivity) to predication.

The disadvantages of wish branches compared to conventional predication are:

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

4.4 Methodology

Figure 4.7 illustrates the simulation infrastructure. We chose the IA-64 ISA to evaluate the wish branch mechanism, because of its full support for predication, but we converted the IA-64 instructions to micro-operations (μops) to execute on our out-of-order superscalar processor model. We modified the ORC compiler [57] to generate the IA-64 binaries (with and without wish branches). The binaries were then run on an Itanium II machine using the Pin binary instrumentation tool [49] to generate traces. These IA-64 traces were later converted to μops. The μops were fed into a cycle-accurate simulator to obtain performance results.

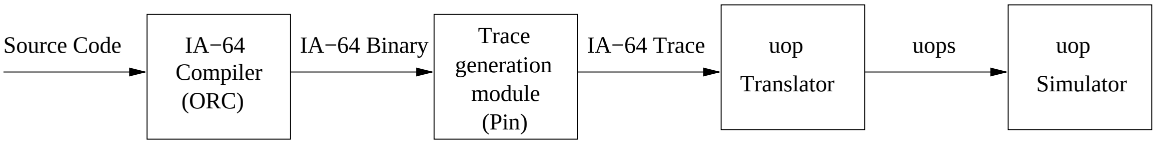


Figure 4.7: Simulation infrastructure

4.4.1 μop Translator and Simulator

We developed an IA-64 translator which converts the disassembled IA-64 instructions into our simulator's native μops. We model μops to be close to a generic RISC ISA. Our translator handles correctly all the issues related to IA-64 specific features such as rotating registers. All NOPs are eliminated during μop translation.

μops are fed into our cycle-accurate simulator. Our baseline processor is an aggressive superscalar, out-of-order processor based on the HPS processor [59, 60]. Table 5.3 describes our baseline microarchitecture. Because a less accurate branch predictor would provide more opportunity for wish branches, a very large and accurate hybrid branch predictor [82, 83, 9] is used in our experiments to avoid inflating the impact of wish branches.

4.4.2 Compilation

All benchmarks were compiled for the IA-64 ISA with the -O2 optimization by the ORC compiler. Software pipelining, speculative loads, and other IA-64 specific optimizations were turned off to reduce the effects of features that are specific to the IA-64 ISA and that are less relevant to an out-of-order microarchitecture. Software pipelining was shown to provide less than 1% performance benefit on the SPEC CPU2000 INT benchmarks [16]

Table 4.2: Baseline processor configuration

Front End	64KB, 4-way, 2-cycle I-cache; 8-wide fetch/decode/rename Fetches up to 3 cond. branch but fetch ends at the first taken branch I-cache stores IA-64 instructions; decoder/ROM produces μops
Branch Predictors	64K-entry gshare [54]/PAs [83] hybrid, 64K-entry selector 4K-entry BTB; 64-entry RAS; 64K-entry indirect target cache minimum branch misprediction penalty is 30 cycles
Execution Core	512-entry reorder buffer; 8-wide execute/retire
On-chip Caches	L1 data cache: 64KB, 4-way, and 2-cycle latency L2 unified cache: 1MB, 8-way, 8 banks, 6-cycle latency All caches use true LRU replacement and have 64B line size
Buses and Memory	300-cycle minimum memory latency; 32 memory banks 32B-wide core-to-memory bus at 4:1 frequency ratio
Predication support	Converted into C-style conditional expressions [74]
Confidence estimator	1KB, tagged (4-way), 16-bit history enhanced JRS estimator [35, 30]

and we removed this optimization to simplify our analysis. Wish branch code generation is also performed with -O2 optimization. To compare wish branches to normal branches and predication, we generated five different binaries for each benchmark, which are described in Table 4.3. Unless otherwise noted, all execution time results reported in this chapter are normalized to the execution time of the normal branch binaries. Section 4.4.2.1 and 4.4.2.2 briefly describe the compilation algorithms we use in our experiments.

4.4.2.1 Predicated Code Binary Generation Algorithm

Figure 4.8 shows the major phase ordering in code generation of ORC. ORC does a region based compilation [48]. Hence, the compiler forms a region first to perform all the optimizations in a region boundary. If-conversion is in one of the early phases, since after if-conversion, the compiler can do other optimizations.

To generate predicated code, the ORC compiler first checks whether or not the

Table 4.3: Description of binaries compiled to evaluate the performance of different combinations of wish branches

Binary name	Branches that can be predicated with the ORC algorithm [48, 57, 53] ...	Backward branches...
normal branch binary	remain as normal branches	remain as normal branches
predicated code binary: BASE-DEF	are predicated based on the compile-time cost-benefit analysis	remain as normal branches
predicated code binary: BASE-MAX	are predicated	remain as normal branches
wish jump/join binary	are converted to wish jumps/joins or are predicated	remain as normal branches
wish jump/join/loop binary	are converted to wish jumps/joins or are predicated	are converted to wish loops or remain as normal branches

control-flow graph is suitable for if-conversion in a region boundary. The ORC compiler performs if-conversion within a region boundary. When the control-flow graph is suitable for if-conversion, the compiler calculates the following equations. Each probability in these equations is determined using compiler heuristics. Execution times are estimated with dependency height and resource usage analysis. We set the branch misprediction penalty to 30 cycles. In the BASE-DEF binary, branches which satisfy Equation (4.3) are converted to predicated code. In the BASE-MAX binary, all branches that are suitable for if-conversion are converted to predicated code. Hence, the BASE-MAX binary contains code that is more aggressively predicated. We use two predicated code binaries as our baselines because neither binary performs the best for all benchmarks. For some benchmarks BASE-DEF performs better and for others BASE-MAX performs better.

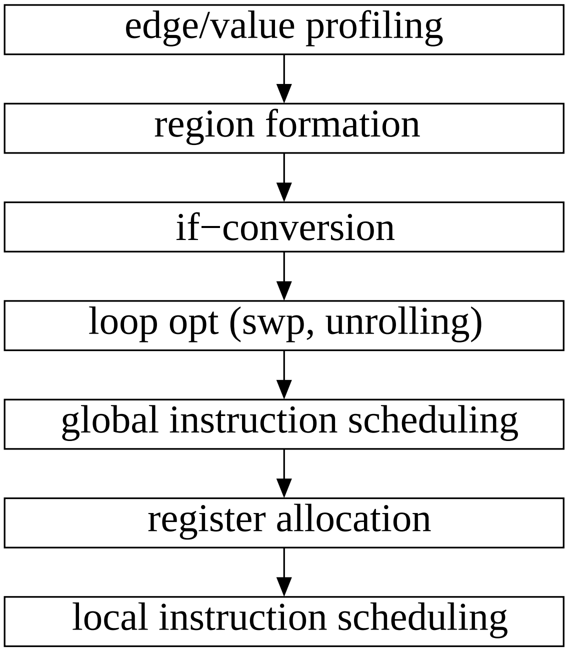

edge/value profiling
region formation
if−conversion
loop opt (swp, unrolling)
global instruction scheduling
register allocation
local instruction scheduling

Figure 4.8: Major phase ordering in code generation of the ORC compiler [38]

$$\begin{aligned} Exec.\,time\,of\,normal\,branch\,code &= exec\_T * P(T) + exec\_N * P(N) + \\ &\quad misp\_penalty * P(misprediction), \quad (4.1) \\ Exec.\,time\,of\,predicated\,code &= exec\_pred, \quad (4.2) \\ Exec.\,time\,of\,predicated\,code &< Exec.\,time\,of\,normal\,br.\,code, \quad (4.3) \end{aligned}$$

where

$exec\_T$: Exec. time of the code when the br. under consideration is taken,

$exec\_N$: Exec. time of the code when the br. under consideration is not taken,

$P(case)$: The probability of the case; e.g., P(T) is the prob. that the br. is taken,

$misp\_penalty$: Machine-specific branch misprediction penalty, and

$exec\_pred$: Execution time of the predicated code.

4.4.2.2 Wish Branch Binary Generation Algorithm

Figure 4.9 shows the modified code generation phases in ORC to generate wish branches. Specifically the if-conversion and loop optimization phases (shaded boxes in Figure 4.9) are modified to generate wish branches.

If a branch is suitable for if-conversion, we treat that branch as a wish branch candidate. If the number of instructions in the fall-through block of a branch is greater than N (we set N to 5), the candidate branch is converted to a wish jump and the necessary wish joins are inserted. Otherwise, the wish branch candidate is converted to predicated code. We use a threshold of 5 instructions because we have found that very short forward branches are better off being predicated. A loop branch is converted to a wish loop if the number of instructions in the loop body is less than L (we set L to 30). We have not tuned the thresholds N and L used in these heuristics. Since our baseline compiler is not optimized to build large predicated code blocks, we inserted some of the wish branches using a binary instrumentation tool when the control flow is suitable to be converted to wish branch code.

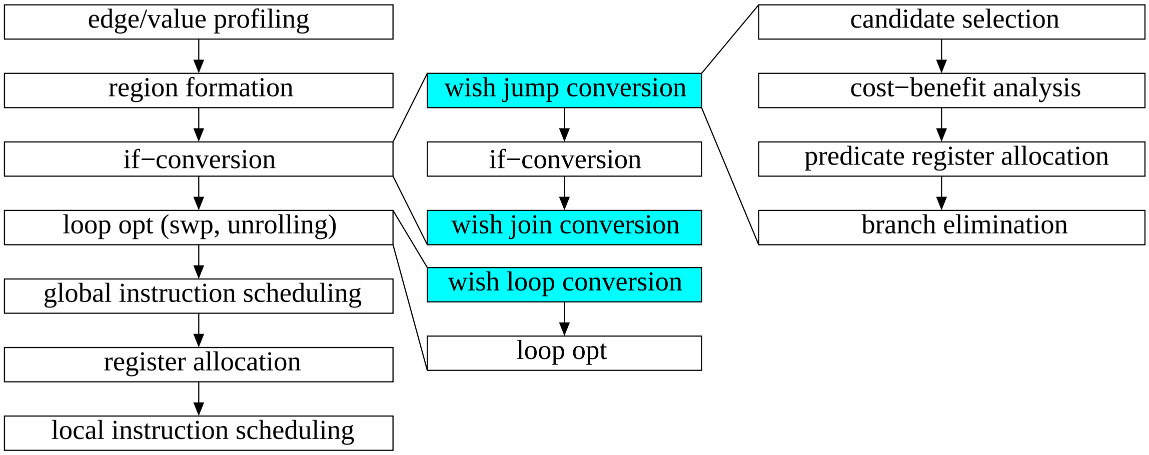


Figure 4.9: Modified code generation phases

4.4.3 Trace Generation and Benchmarks

IA-64 traces were generated with the Pin instrumentation tool [49]. Because modeling wrong-path instructions is important in studying the performance impact of wish branches, we generated traces that contain wrong-path information by forking a wrong-path trace generation thread. We forked a thread at every wish branch down the mispredicted path. The spawned thread executed until the number of executed wrong-path instructions exceeded the instruction window size. The trace contains the PC, predicate register, register value, memory address, binary encoding, and the current frame marker information for each instruction.

All experiments were performed using the SPEC INT 2000 benchmarks. The benchmarks were run with a reduced input set [46] to simulate until the end of the program. The information about the simulated benchmarks for the normal branch binaries and the wish jump/join/loop binaries are shown respectively in Table 4.4 and Table 4.5.[4]

[4] Due to problems encountered during trace generation using Pin, gcc, perlbmk and eon benchmarks were excluded. NOPs are included in the dynamic IA-64 instruction count, but they are not included in the μop

Branch information displayed is collected only for conditional branches. For the wish jump/join/loop binaries, we show the total number of static and dynamic wish branches and the percentage of wish loops among all wish branches.

Table 4.4: Simulated benchmarks: characteristics of normal branch binaries

Benchmark	Dynamic instructions IA64 instructions / μops	Static branches	Dynamic branches	Mispredicted branches (per 1000 μops)	IPC/μPC
164.gzip	303M / 211M	1271	31M	8.3	2.25/ 1.53
175.vpr	161M / 106M	4078	13M	7.8	2.38/ 1.60
181.mcf	189M / 135M	1288	28M	4.7	1.52/ 1.46
186.crafty	316M / 227M	4334	30M	4.7	1.68/ 1.01
197.parser	428M / 311M	2879	72M	9.6	1.21/ 0.87
254.gap	611M / 423M	4163	50M	1.0	1.22/ 0.80
255.vortex	113M / 87M	7803	12M	0.8	1.06/ 0.84
256.bzip2	429M / 308M	1236	40M	8.6	1.38/ 1.37
300.twolf	171M / 114M	4306	10M	6.8	1.81/ 1.16

Table 4.5: Simulated benchmarks:characteristics of wish branch binaries

Benchmark	Static wish branches (% of wish loops)	Dynamic wish branches (% of wish loops)
164.gzip	93 (80%)	9.5M (61%)
175.vpr	206 (83%)	4.3M (35%)
181.mcf	31 (54%)	5.1M (20%)
186.crafty	271 (65%)	3.7M (49%)
197.parser	214 (88%)	14.2M (63%)
254.gap	167 (74%)	6.1M (75%)
255.vortex	104 (33%)	1.7M (62%)
256.bzip2	130 (81%)	8.7M (90%)
300.twolf	356 (71%)	3.1M (57%)

count.

4.5 Simulation Results and Analysis

4.5.1 Wish Jumps/Joins

We first evaluate how using wish jumps/joins performs compared to normal branches and predicated code. Figure 4.10 shows the normalized execution time of four different configurations for each benchmark: (1) BASE-DEF binary, (2) BASE-MAX binary, (3) wish jump/join binary with a real confidence estimator, and (4) wish jump/join binary with a perfect confidence estimator. With a real confidence estimator, the wish jump/join binaries improve the average execution time by 11.5% over the normal branch binaries and by 10.7% over the best-performing (on average) predicated code binaries (BASE-DEF). The wish jump/join binaries perform better than the normal branch binaries for all the benchmarks, except mcf. Moreover, they perform better than both of the predicated code binaries for gzip, vpr, mcf, gap, and, twolf. For vpr, mcf, and twolf, three benchmarks where the overhead of predicated execution is very high, as was shown in Figure 2.5, the wish jump/join binaries improve the execution time by more than 10% over the predicated code binaries. Note that, the execution time of mcf skews the average normalized execution time, because mcf performs very poorly with predicated execution. Hence, this chapter reports two average execution time numbers on the graphs. The set of bars labeled AVG shows the average execution time with mcf included. The set of bars labeled AVGnomcf shows the average execution time with mcf excluded.

Figure 4.10 also shows that the wish jump/join binaries reduce the overhead which causes the predicated code binaries to perform worse than the normal branch binaries. For example, the BASE-DEF binaries perform worse than the normal branch binaries for gzip, mcf, crafty, and gap. Similarly, the BASE-MAX binaries perform worse than the normal branch binaries on mcf and bzip2. In fact, aggressive predication (BASE-MAX) increases the execution time of mcf by 102% because of the additional delay caused by predicated instructions. In mcf, the execution of many critical load instructions that would cause

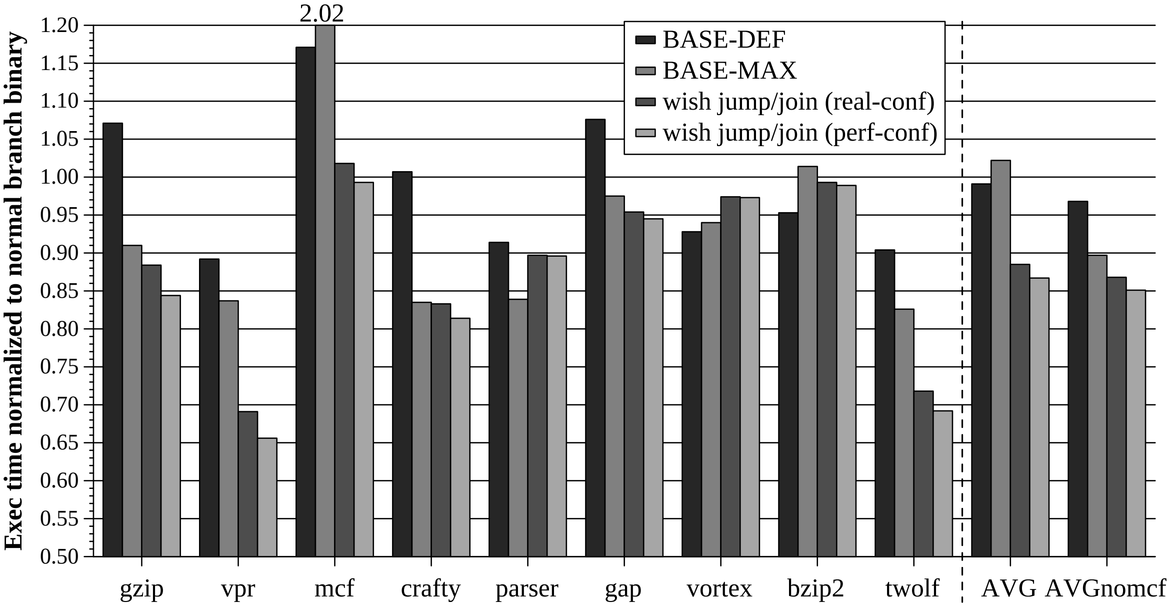


Figure 4.10: Performance of wish jump/join binaries

cache misses are delayed because their source predicates are dependent on other critical loads which incur cache misses. Hence, predicated execution results in the serialization of many critical load instructions that would otherwise be serviced in parallel had branch prediction been used, leading to a large performance degradation. The wish jump/join binaries eliminate the performance loss due to predicated execution on benchmarks where predicated execution reduces performance. Hence, wish branches are effective at reducing the negative effects of predicated execution.

The wish jump/join binary performs worse than both of the predicated code binaries only for one benchmark, vortex. This is due to the reduced size of the basic blocks in the wish jump/join binary for vortex. The compiler is able to optimize the code better and more aggressively in the predicated code binaries that have larger basic blocks. Note that the compiler heuristics we used to insert wish branches are very simple. Better heuristics that take into account more information, as explained in Section 4.2.2.1, can eliminate the disadvantages caused by wish branches in vortex.

Figure 4.11 shows the dynamic number of wish branches per 1 million retired μops. The left bar for each benchmark shows the number of wish branches predicted to have low-confidence and how many of those were mispredicted. The right bar shows the number of wish branches predicted to have high-confidence and how many of those were mispredicted. Ideally, we would like two conditions to be true. First, only the actually mispredicted wish branches should be estimated as low-confidence. Second, no mispredicted wish branch should be estimated as high-confidence. Figure 4.11 shows that the second condition is much closer to being satisfied than the first on all benchmarks. Very few of the high-confidence branches are actually mispredicted. However, the first condition is far from being satisfied, especially in gzip, vpr, mcf, crafty, and twolf. In these benchmarks, a significant number of wish branches are estimated as low-confidence even though they are not mispredicted.[5] Therefore, a better confidence estimator would improve the performance of wish branches on these benchmarks, as shown in the rightmost bars in Figure 4.10.

Figure 4.11 also provides insight into why wish branches improve the performance of predicated execution significantly in some benchmarks. For example, in mcf most of the branches that are converted to wish branches are correctly predicted. These branches are predicated in the BASE-MAX binary. However, predicating them reduces the performance with the reduced input set, because those branches are almost always correctly predicted. Converting them into wish branches rather than predicating them allows the hardware to dynamically decide whether or not they should be predicated. As shown in Figure 4.11, the hardware confidence estimator does well on mcf and correctly identifies most of the correctly-predicted wish branches as high-confidence. Hence, for those wish branches, the overhead of predicated execution is avoided and the wish branch binary performs as well

[5] As Jiménez and Lin discussed in [37], a confidence estimator usually has high a coverage with a low accuracy or a low coverage with a high accuracy.

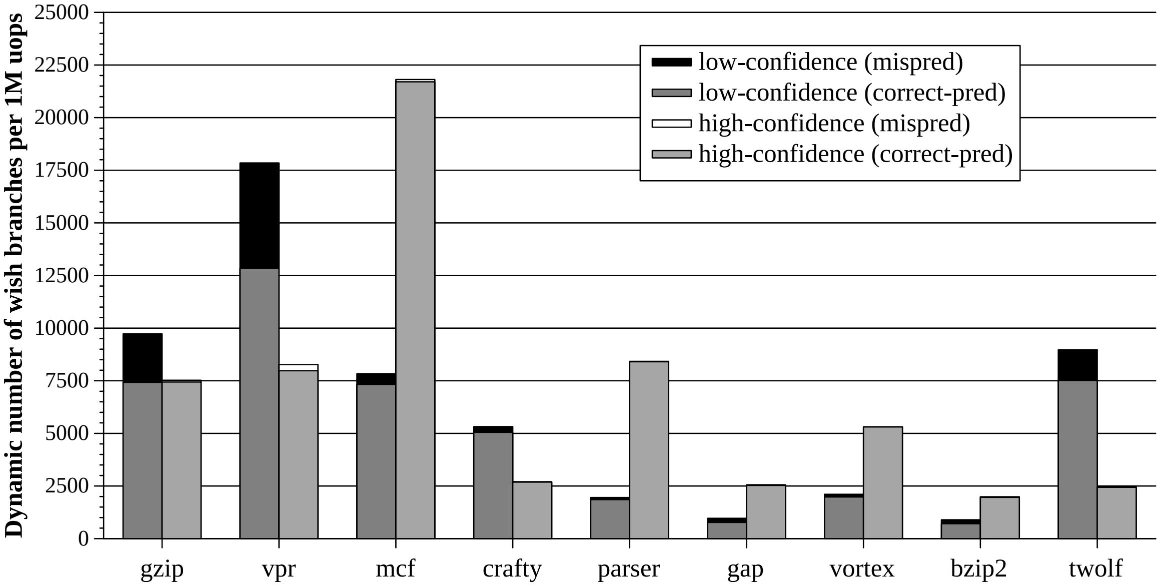


Figure 4.11: Dynamic number of wish branches per 1M retired μops. Left bars: low-confidence, right bars: high-confidence.

as the normal branch binary. Similarly in gzip, vpr, and gap, many of the wish branches are correctly predicted and also estimated as high confidence, resulting in significant savings in the overhead of predicated execution, which is reflected in the performance of the wish jump/join binaries for these three benchmarks in Figure 4.10. Most wish branches are correctly predicted and identified as high-confidence also in parser and vortex. However, the performance of parser and vortex is not improved with wish branches compared to the predicated code binaries because the overhead of predicated execution is very low for these two benchmarks as shown in Figure 2.5.

4.5.2 Wish Jumps/Joins and Wish Loops

Figure 4.12 shows the performance of wish branches when wish loops are also used in addition to wish jumps/joins. With a real confidence estimator, the wish jump/join/loop binaries improve the average execution time by 14.2% compared to the normal branch binaries and by 13.3% compared to the best-performing (on average) predicated code binaries (BASE-DEF). An improved confidence estimator has the potential to increase the performance improvement up to 16.2% compared to the normal branch binaries. Even if mcf is excluded from the calculation of the average execution time, the wish jump/join/loop binaries improve the average execution time by 16.1% compared to the normal branch binaries and by 6.4% compared to the best-performing predicated binaries (BASE-MAX), with a real confidence estimator.

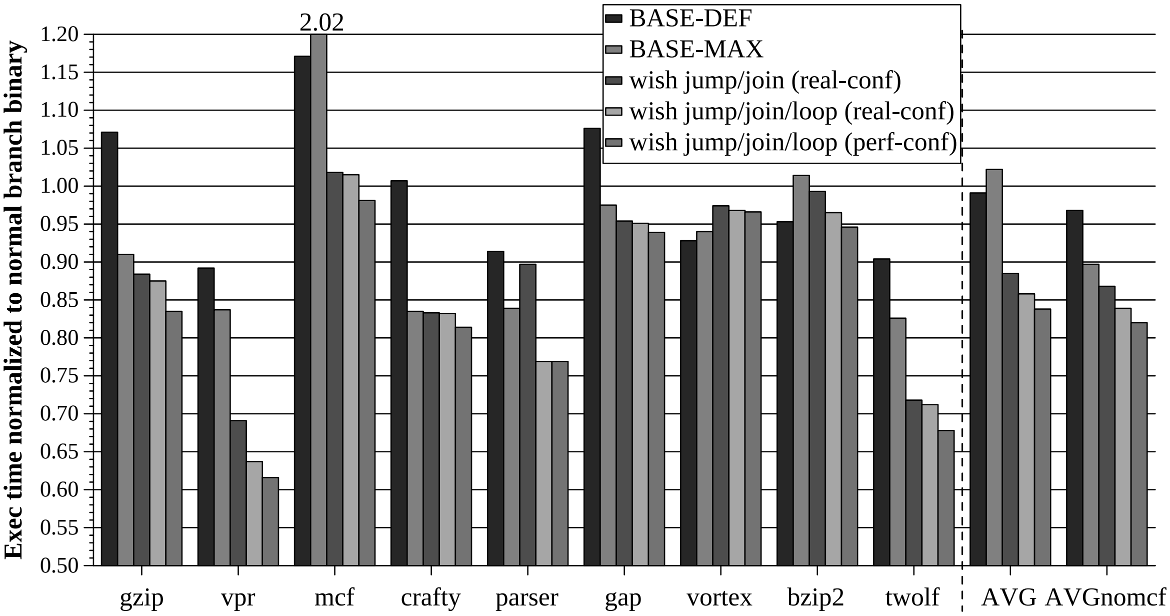


Figure 4.12: Performance of wish jump/join/loop binaries

Using wish loops in addition to wish jumps/joins improves the execution time of vpr, parser, and bzip2 by more than 3%. The reason for the performance improvement on these three benchmarks can be seen in Figure 4.13. This figure shows the dynamic number of wish loops per 1 million μops and classifies them based on their confidence estimation and misprediction status. Remember that the *late-exit* misprediction case is the only case where a wish loop improves performance compared to a normal loop branch, as described in Section 4.1.2. In vpr, parser, and bzip2 there is a significant number of wish loop instructions that are predicted to be low-confidence and are actually mispredicted as *late-exit*. Therefore, we see significant performance improvements due to wish loops for these benchmarks.

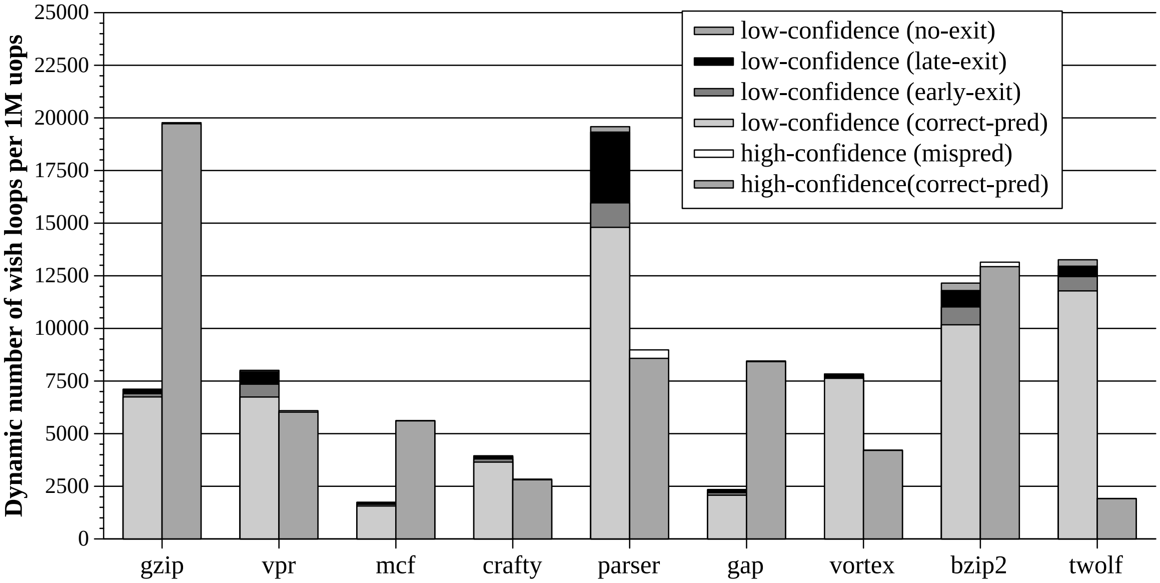


Figure 4.13: Dynamic number of wish loops per 1M retired μops. Left bars: low-confidence, right bars: high-confidence.

4.5.2.1 Source Code Example for Wish Loops

Wish loops provide significant performance benefit for the parser benchmark. Figure 4.14 shows the high level source code of one of the major wish loops in this benchmark. This function checks where a period symbol is inside a given word. Since an English word has usually fewer than 16 characters, the number of iterations of the loop is usually fewer than 16. However, a lot of abbreviations have a period symbol after the first character. Due to abbreviations, the frequency of the loop iterations over the whole run of the benchmark shows a high peak for 1 iteration as shown in Figure 4.15. The frequency of the loop also has a normal distribution with a mean of 6-7 iterations. In this example, the number of iterations is very unpredictable (because it is dependent on the input word) but it is more likely to be smaller than 16. Therefore, this loop branch is a very good wish loop candidate.

```
int numberfy(char *c)
{
for (; (*s !=\0)&& (*s != .);s++) ;  // the for loop becomes a wish loop branch
...
}
```

Figure 4.14: An example from parser showing an loop branch

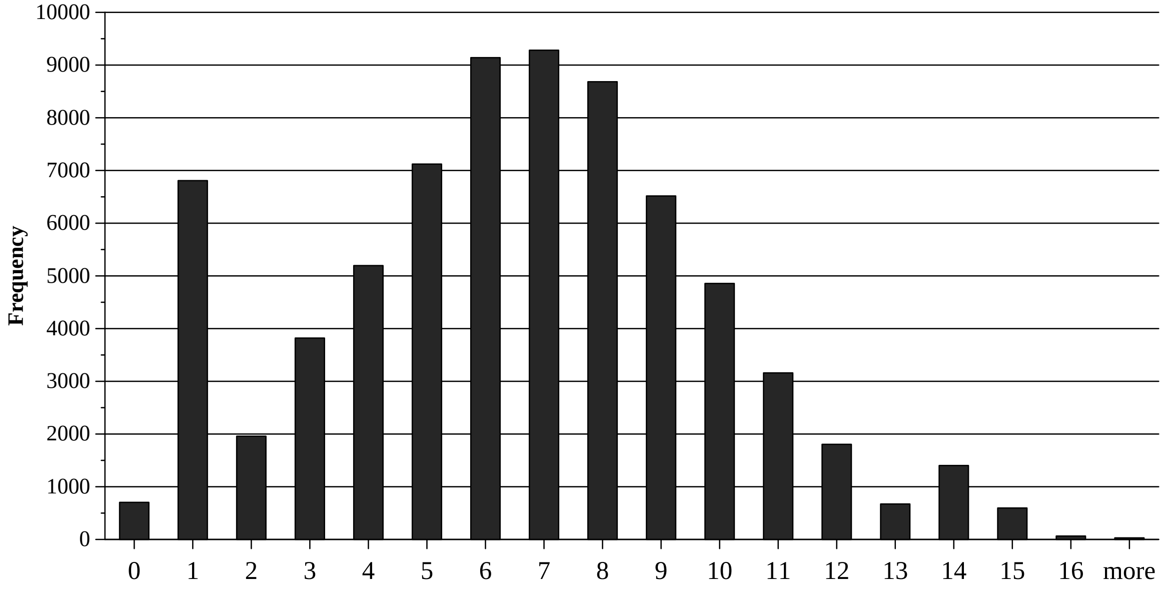


Figure 4.15: Frequency of loop iteration of for the branch in Figure 4.14

4.5.3 Comparisons with the Best-Performing Binary for Each Benchmark

We also compare the performance of wish branches to the best-performing binary for each benchmark. To do so, we selected the best-performing binary for each benchmark among the normal branch binary, BASE-DEF predicated code binary, and BASE-MAX predicated code binary based on the execution times of these three binaries, which are obtained via simulation. Note that this comparison is unrealistic because it assumes that the compiler can, at compile-time, predict which binary would perform the best for the benchmark at run-time. This assumption is not correct, because the compiler does not know the run-time behavior of the branches in the program. Even worse, the run-time behavior of the program can also vary from one run to another run. Hence, depending on the input set to the program, a different binary could be the best-performing binary, as we have already shown in Figure 1.1.

Table 4.6: Execution time reduction of the wish jump/join/loop binaries over the best-performing binaries on a per-benchmark basis (using the real confidence mechanism). DEF, MAX, BR (normal branch) indicate which binary is the best performing binary for a given benchmark.

	column 1	column 2		column 3	
Benchmark	% exec time reduction vs. normal branch binary	% exec time reduction vs. the best predicated code binary for the benchmark		% exec time reduction vs. the best non-wish-branch binary for the benchmark	
gzip	12.5%	3.8%	MAX	3.8%	MAX
vpr	36.3%	23.9%	MAX	23.9%	MAX
mcf	-1.5%	13.3%	DEF	-1.5%	BR
crafty	16.8%	0.4%	MAX	0.4%	MAX
parser	23.1%	8.3%	MAX	8.3%	MAX
gap	4.9%	2.5%	MAX	2.5%	MAX
vortex	3.2%	-4.3%	DEF	-4.3%	DEF
bzip2	3.5%	-1.2%	DEF	-1.2%	DEF
twolf	29.8%	13.8%	MAX	13.8%	MAX
AVG	14.2%	6.7%		5.1%	

Table 4.6 shows, for each benchmark, the reduction in execution time achieved with the wish jump/join/loop binary compared to the normal branch binary (column 1), the best-performing predicated code binary for the benchmark (column 2), and the best-performing binary (that does not contain wish branches) for the benchmark (column 3). Even if the compiler were able to choose and generate the best-performing binary for each benchmark, the wish jump/join/loop binary outperforms the best-performing binary for each benchmark by 5.1% on average, as shown in the third column.

4.5.4 Sensitivity to Microarchitectural Parameters

4.5.4.1 Effect of the Instruction Window Size

Figure 4.16 shows the normalized execution time of the wish jump/join/loop binaries on three different machines with 128, 256, and 512-entry instruction windows. The data shown in the left graph is averaged over all the benchmarks examined. The data in the right graph is averaged over all benchmarks except mcf. The execution time of each binary is normalized to the execution time of the normal branch binary on the machine with the corresponding instruction window size. Compared to the normal branch binaries, the wish jump/join/loop binaries improve the execution time by 11.4%, 13.0%, and 14.2% respectively on a 128, 256, and 512-entry window processor. Wish branches provide larger performance improvements on processors with larger instruction windows. This is due to the increased cost of branch mispredictions (due to the increased time to fill the instruction window after the pipeline is flushed) on machines with larger instruction windows. Wish loops are also more effective on larger windows, because, with a larger window, it is more likely that the front-end of the processor has already exited the loop when a mispredicted wish loop branch is resolved. This increases the likelihood of the late-exit case.

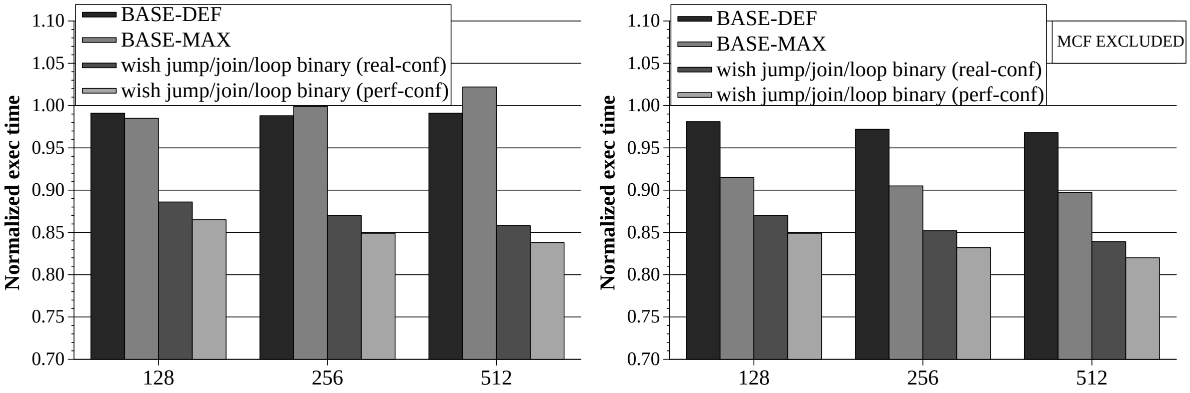


Figure 4.16: Effect of instruction window size on wish branch performance. The left graph shows the average execution time over all benchmarks, the right graph shows the average execution time over all benchmarks except mcf.

4.5.4.2 Effect of the Pipeline Depth

Figure 4.17 shows the normalized execution time of the five binaries on three different 256-entry window processors with 10, 20, and 30 pipeline stages. Compared to the normal branch binaries, the wish jump/join/loop binaries improve the execution time by 8.0%, 11.0%, and 13.0% respectively on processors with 10, 20, and 30 pipeline stages. The performance benefits of wish branches increase as the pipeline depth increases, since the branch misprediction penalty is higher on processors with deeper pipelines. The wish jump/join/loop binaries always significantly outperform the normal branch and predicated code binaries for all pipeline depths and instruction window sizes examined.

4.5.4.3 Effect of the Mechanism Used to Support Predicated Execution

Our baseline out-of-order processor uses C-style conditional expressions to handle predicated instructions as described in Section 2.2.1. We also implemented the select-μop mechanism proposed by Wang et al. [79](Section 2.2.3) to quantify the benefits of wish branches on an out-of-order microarchitecture that uses a different technique to support

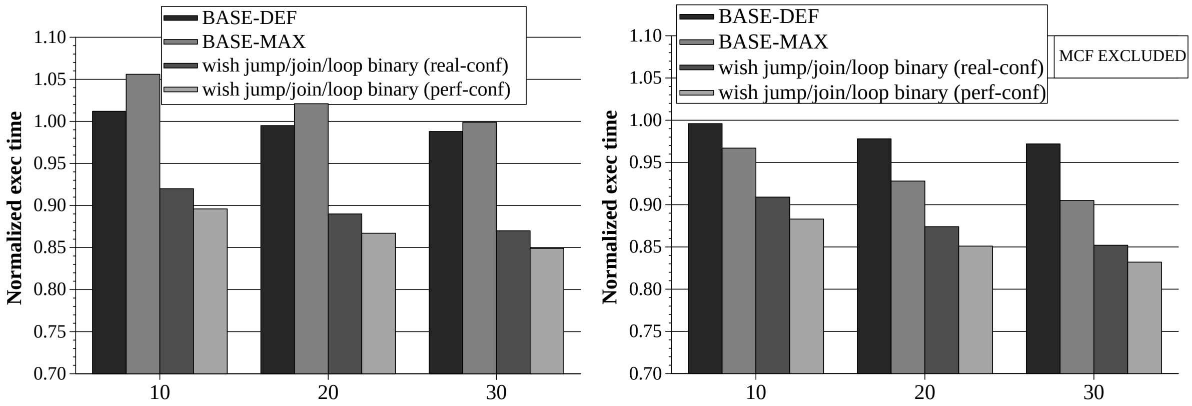


Figure 4.17: Effect of pipeline depth on wish branch performance.

predicated execution.

The advantage of the select-μop mechanism over the C-style conditional expressions is that it does not require the extra register read port and the extra input in the datapath to read and carry the old destination register value. Hence, the implementation cost of predicated execution is lower on a processor that supports predicated instructions using the select-μop mechanism. The select-μop also enables the execution of a predicated instruction before its source predicate value is ready, but the dependents of the predicated instruction still cannot be executed until the source predicate is resolved. Since dependent instructions cannot be executed, we found that a significant portion of the overhead of predicated execution still remains on a processor implementing the select-μop mechanism.

The disadvantage of the select-μop mechanism is that it requires additional μops to handle the processing of predicated instructions. Note that this is not the case in a processor that supports predicated instructions using C-style conditional expressions. Due to this additional μop overhead, the performance benefits of predicated code are lower on a processor that uses the select-μop mechanism than on a processor that uses C-style conditional expressions.

Figure 4.18 shows the normalized execution time of the predicated code, wish jump/join, and wish jump/join/loop binaries on a processor that supports predicated execution using the select-μop mechanism. With a real confidence estimator, the wish jump/join/loop binaries improve the average execution time by 11.0% compared to the normal branch binaries and by 14.0% compared to the best-performing (on average) predicated code binaries (BASE-DEF). On the processor that uses the select-μop mechanism, the overall performance improvement of wish branches over conditional branch prediction (11.0%) is smaller than it is on the processor that uses C-style conditional expressions (14.2%). This is due to the higher instruction overhead of the select-μop mechanism to support the predicated instructions. On the other hand, the overall performance improvement of wish branches over predicated execution (14.0%) is larger than it is on the processor that uses C-style conditional expressions (13.3%). Hence, the performance benefit of wish branches over predicated execution is larger when predicated execution has higher overhead.

4.5.4.4 Wish Branches in In-Order Processors

We also evaluate the benefits of wish branches in an in-order machine. The processor we evaluate has a 30-cycle minimum branch misprediction penalty. Since branch mispredictions are less costly on an in-order machine, predicated code binaries do not show performance benefits as large as they do on out-of-order machines. Even so, wish branches still reduce most of the negative effects of predicated code and keep the benefits of predicated code if the predicated code provides a performance benefit. Wish jump/join/loop binaries improve the performance of the in-order processor by 6.0% compared to traditional conditional branches and by 1% compared to BASE-MAX. But compared to BASE-DEF, wish jump/join/loop binaries reduce performance by 2.1%. In an in-order processor, BASE-MAX binaries perform worse than BASE-DEF on gzip, vpr, mcf, vortex, and bzip2. Unlike an out-of-order processor, an in-order processor is less tolerant of the increasing

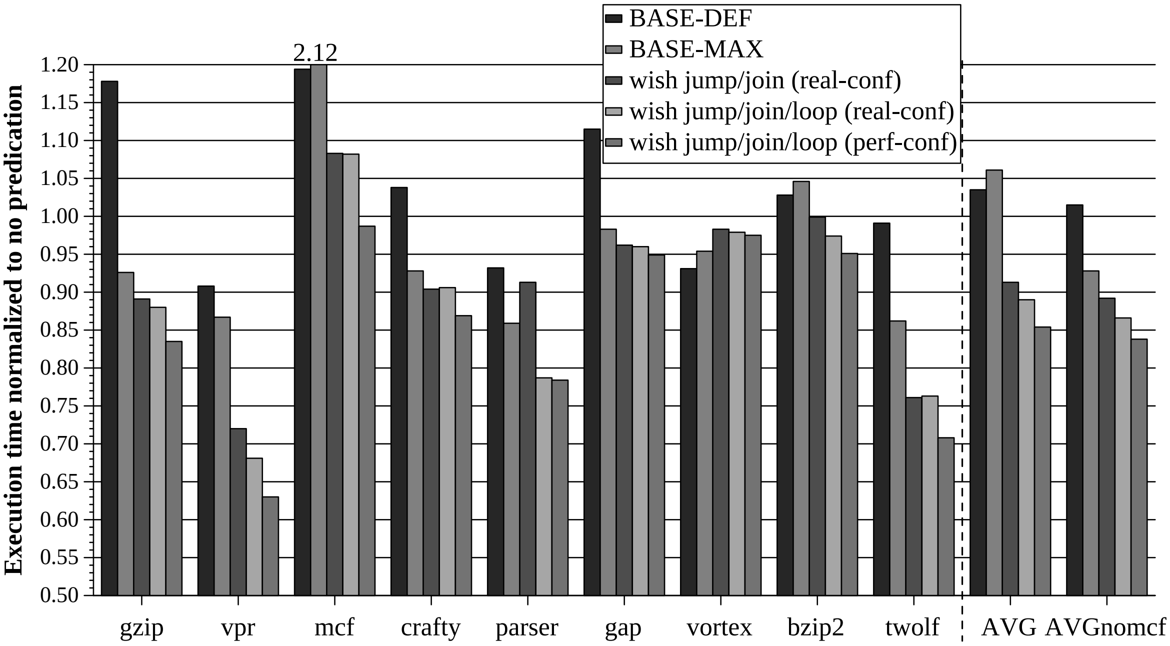


Figure 4.18: Performance of wish branches on an out-of-order processor that implements the select-μop mechanism

number of executed instructions. BASE-MAX binaries have a higher number of instructions than BASE-DEF binaries because more branches are converted to predicated code in BASE-MAX binaries. Since wish branch binaries also have a higher number of instructions than BASE-DEF binaries, wish branch binaries sometimes perform worse than BASE-DEF binaries. If we are to consider using wish branches in an in-order processor, the cost-benefit analysis to generate wish branches in an in-order processor should be developed.

4.5.4.5 Performance Analysis

Figure 4.20 shows the number of fetched μops for three different binaries. All the results are normalized to normal branch binaries. Since wish branches reduce the number of pipeline flushes, the number of fetched instructions is reduced significantly (14%). Hence wish branches would improve the energy efficiency. Section 5.5.5 will analyze the power

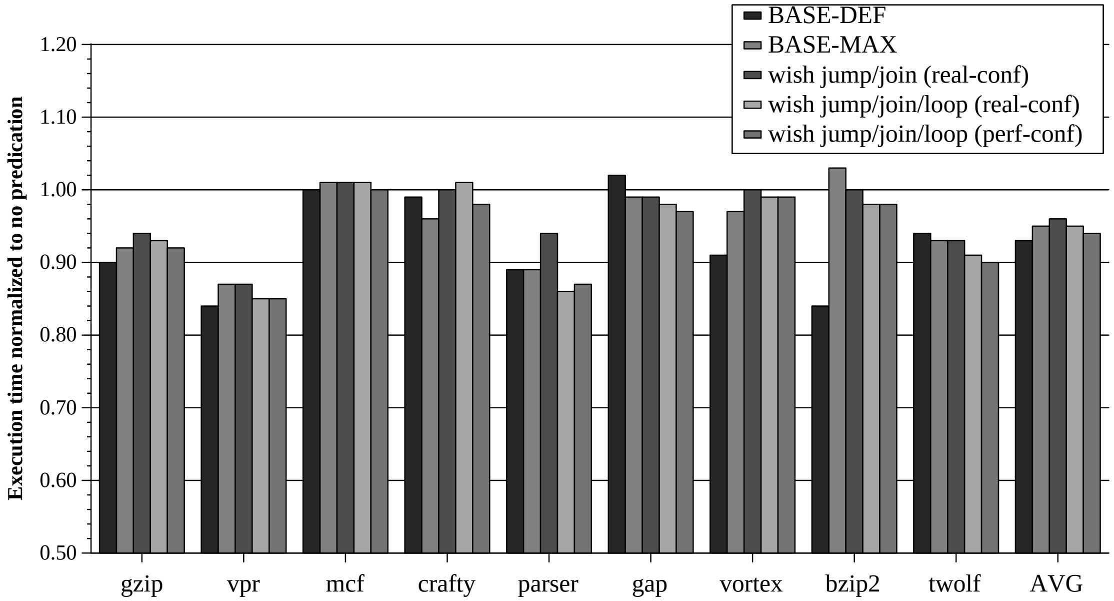


Figure 4.19: Normalized execution time in an in-order processor

and energy consumption of wish branches.

With the BASE-MAX binary, mcf's execution time significantly increases in comparison to other benchmarks. As we discussed in Section 4.5.1, the main reason is the execution delay of memory operations that depend on predicate values. Figure 4.21 shows the result of an ideal experiment. In this experiment, all load operations take two cycles, which is equivalent to the data cache access time. (In other words, we simulate a perfect data cache.) The result shows that most of the performance degradation of mcf in the BASE-MAX binary is eliminated (from 112% to 14%). The performance results for the remaining benchmarks are similar to the results with the baseline with the real data cache. Hence, we conclude that the main reason of mcf's performance degradation in the BASE-MAX binary is the execution delay of predicate-dependent memory operations.

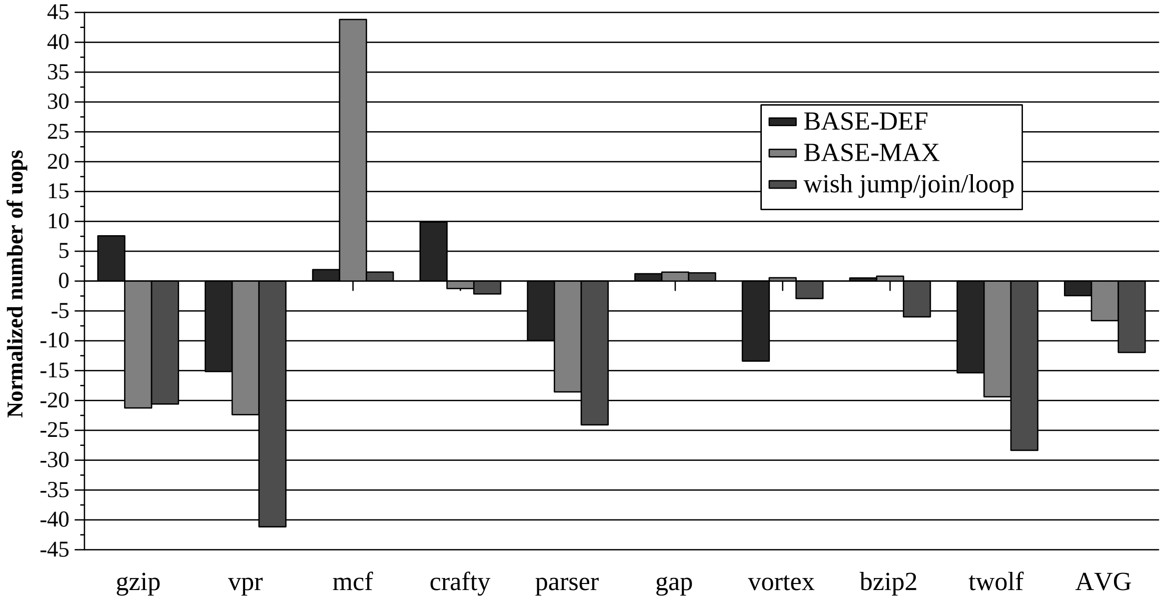


Figure 4.20: The number of fetched μops normalized to non-predicated binaries

4.5.4.6 Effect of Front-end Design

Figure 4.22 shows the performance of four evaluated binaries with different front-end configurations: the maximum number of conditional branches that can be fetched in one cycle is varied from 2 (less aggressive front-end) to 4 (aggressive front-end). All the results are normalized to the execution time of non-predicated binaries in the same machine configuration. The results show that the performance benefit of wish branches is not sensitive to the aggressiveness of the front-end design. All three cases show 14% performance benefit. One of the benefits of predicated code is that it reduces the number of branches, which could increase the average number of instructions fetched in one cycle as compared to normal branch code. (i.e., normal branch code has more fetch breaks.) With wish branches, this benefit of predicated code is lost since wish branch code does not eliminate the predicated branches. However, as shown in Figure 4.22, this effect is not significant. There are two reasons for this. First, as we showed in Section 4.2.2.1, short

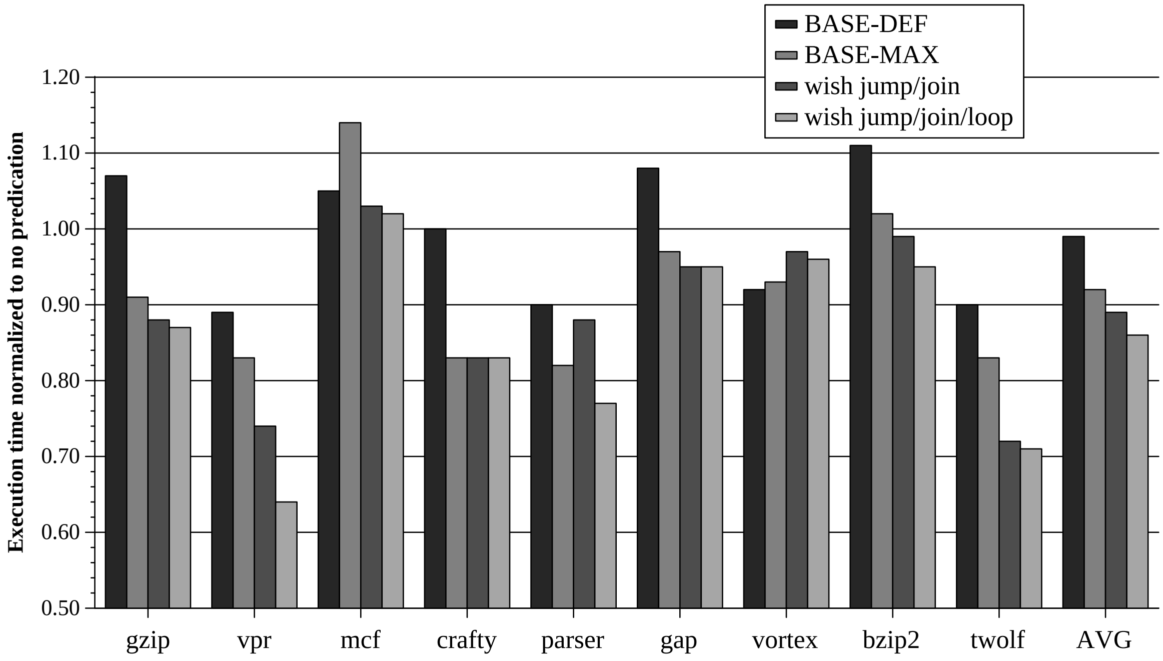


Figure 4.21: Normalized execution time with a perfect D-cache

forward branches are already converted to predicated code in the wish branch binaries. Second, the performance benefit of reducing the branch misprediction penalty and providing adaptivity to predicated execution significantly outweighs the performance loss due to the reduction in the average number of instructions fetched in one cycle. This is because fetch breaks due to limitations on the number of branches that can be fetched in one cycle happens rarely in the baseline processor.

4.5.4.7 Effect of Different Branch Predictors

Figure 4.23 shows the normalized execution time of the wish jump/join/loop binaries with a 59KB (1021 rows and 59-bit history) perceptron branch predictor [36]. The execution time of each binary is normalized to the execution time of the normal branch binary on the same machine with the perceptron branch predictor. Wish branch binaries improve

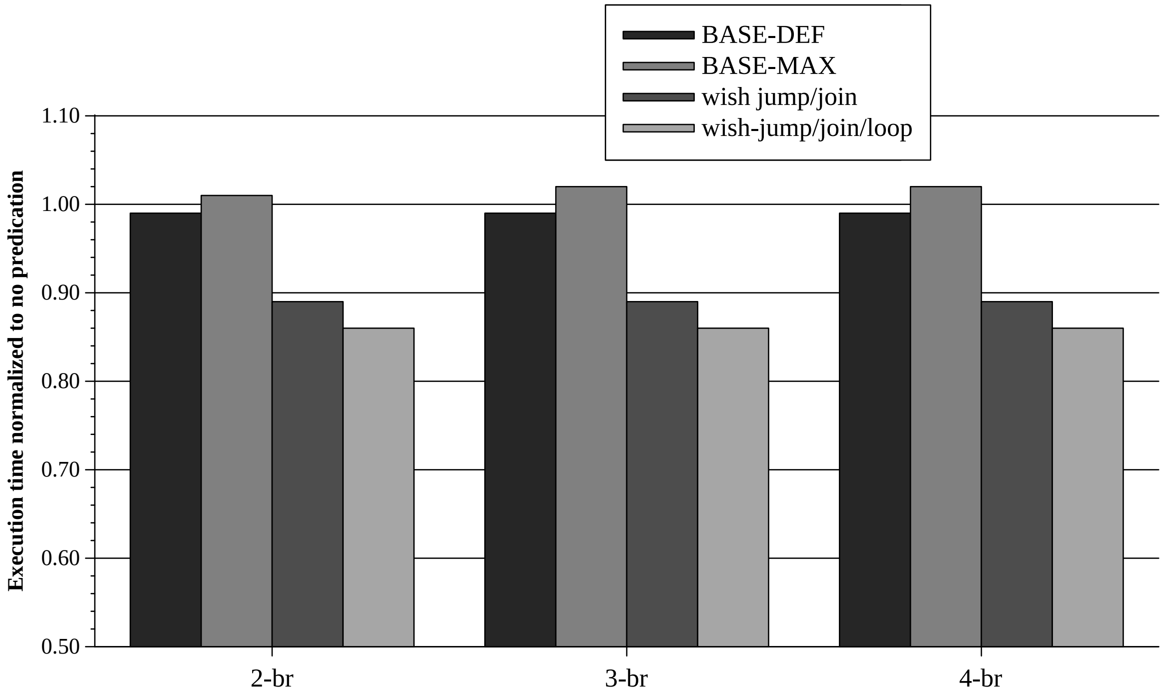


Figure 4.22: Performance of wish branches as a function of the maximum number of conditional branches fetched in a cycle

the average execution time by 13% over the normal branch binaries and by 11.2% over the BASE-DEF binaries. The performance benefit of wish branches is slightly less than the benefit with a 64KB hybrid branch predictor. This is because the perceptron branch predictor provides higher branch prediction accuracy. The average branch prediction accuracy of the perceptron branch predictor is 92.04% and that of the hybrid branch predictor is 91.94%. Furthermore, the confidence estimator used in the evaluation is originally developed for a branch predictor like gshare. However, wish branches still provide significant performance benefit, and their benefit can be further increased with a better confidence estimator tuned for the branch predictor used in the baseline machine.

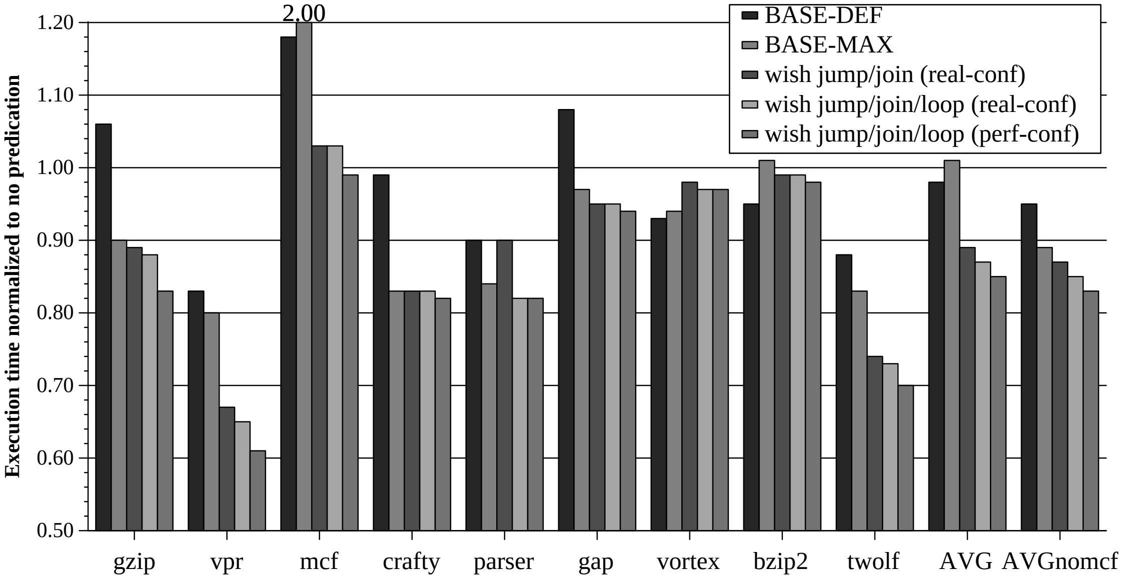


Figure 4.23: Performance of wish branches with a perceptron branch predictor

4.5.5 Comparisons with Predicate Prediction

Section 2.2.4 discussed predicate prediction. A predicate predictor can reduce the problem of execution delay due to dependencies on the predicate values. However, it cannot reduce the problem of increased number of fetched instructions in predicated code. Figure 4.24 shows the performance of BASE-DEF and BASE-MAX binaries with a 512B predicate predictor [19]. Unlike wish branches, which are used selectively, the predicate predictor predicts all predicate values. Although correct predictions of the predicate predictor could improve performance, the results show that the predicate predictor actually reduces performance. The evaluated Predicate predictor has on average 81% prediction accuracy, which means that 19% of predictions cause replay penalty. These results are different from Chuang and Calder [19]'s results. The main reason is, in their work, the baseline machine stalls the pipeline in the rename stage until the multiple definition problem is solved. This stall results in a lot of execution delay. However, our baseline employs the CMOV-style mechanism, so only predicate-value-dependent instructions cannot be executed but the remaining predicate-value-independent instructions can be executed.

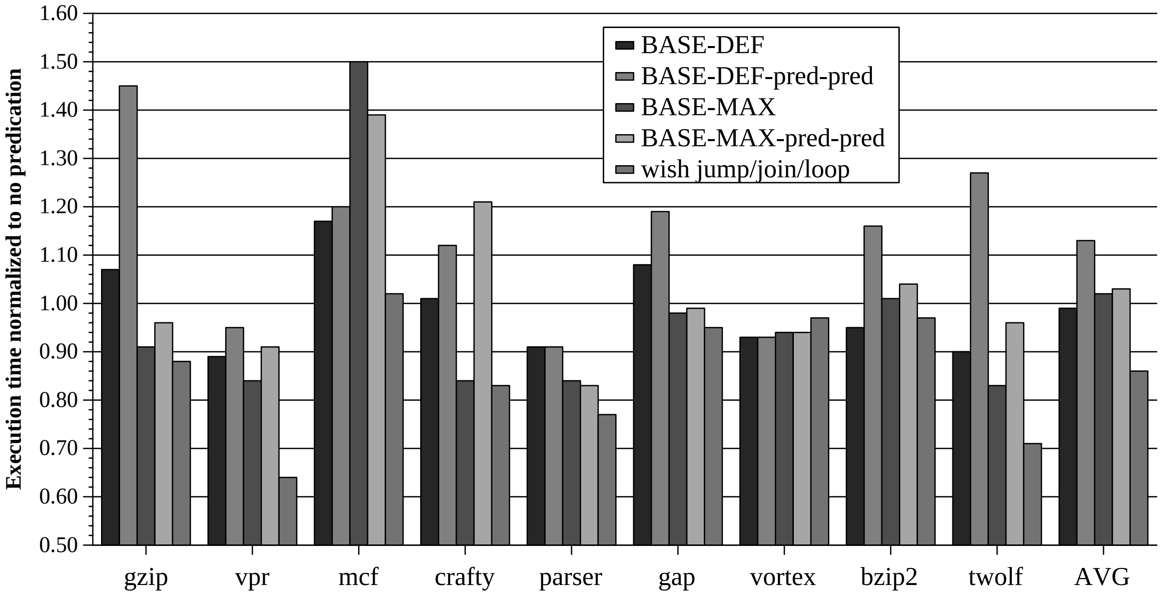


Figure 4.24: Performance with the predicate predictor

Figure 4.25 shows the performance of BASE-DEF and BASE-MAX binaries with a 1KB confidence estimator and a predicate predictor. In this experiment, the processor predicts a predicate value only if the predicate prediction has a high confidence. The confidence estimator estimates the confidence of predicate predictions. If the predicate prediction has a low confidence, the processor does not execute predicated instructions until the predicate value is resolved (i.e., the predicate value is not predicted). The results show that even though using the confidence estimator with the predicate predictor improves performance compared to using only the predicate predictor, predicate prediction still does not perform better than the baseline processor. The reason is that the confidence estimator does not have a good accuracy. With wish branches, the confidence estimator is used only for wish branches. However, with the predicate predictor, the confidence estimator needs to estimate the confidence for all predicate instructions, which results in a significant lower accuracy.

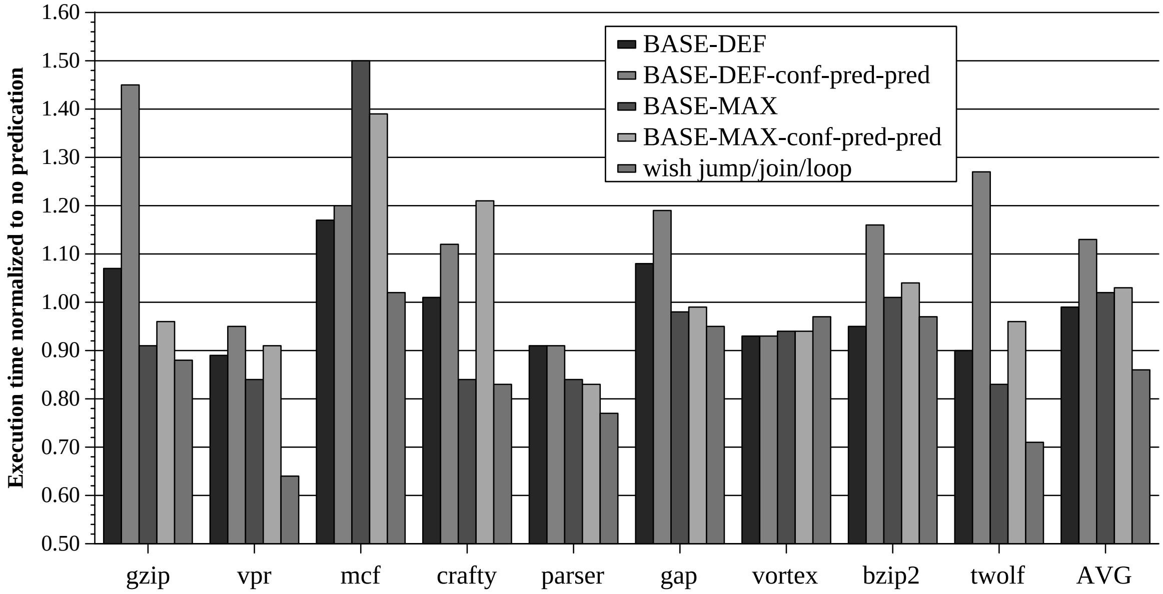


Figure 4.25: Performance with the predicate predictor with a confidence estimator

4.6 Summary

This chapter proposed a new control-flow mechanism called *wish branches* to reduce the negative effects of predicated code and to obtain the best performance of predicated execution and branch prediction. It described the operation of three types of wish branches: wish jumps, wish joins, and wish loops. The major contributions of wish branches to the research in predicated execution and branch misprediction penalty reduction are:

1. Wish jumps and joins provide a mechanism to dynamically eliminate the overhead of predicated execution. These instructions allow the hardware to dynamically choose between using predicated execution versus conditional branch prediction for each dynamic instance of a branch based on the run-time confidence estimation of the branch's prediction.

2. Wish jumps and joins also allow the compiler to generate predicated code more aggressively and using simpler heuristics, since the “bad compile-time decisions” can be corrected at run-time. In previous research, a static branch instruction either remained as a conditional branch or was predicated for *all its dynamic instances*, based on less accurate compile-time information. If the compiler made a bad decision to predicate, there was no way to dynamically eliminate the overhead of the bad decision.
3. Wish loops provide a mechanism to exploit predicated execution to reduce the branch misprediction penalty for *backward* (loop) branches. In previous research, it was not possible to reduce the branch misprediction penalty for a backward branch by solely utilizing predicated execution.

Our results show that wish branches improve the average execution time of nine SPEC INT 2000 benchmarks on an aggressive out-of-order superscalar processor by 14.2% compared to conditional branch prediction only and by 13.3% compared to the best-performing predicated code binary.

Chapter 5

Diverge-Merge Processor (DMP)

5.1 Introduction

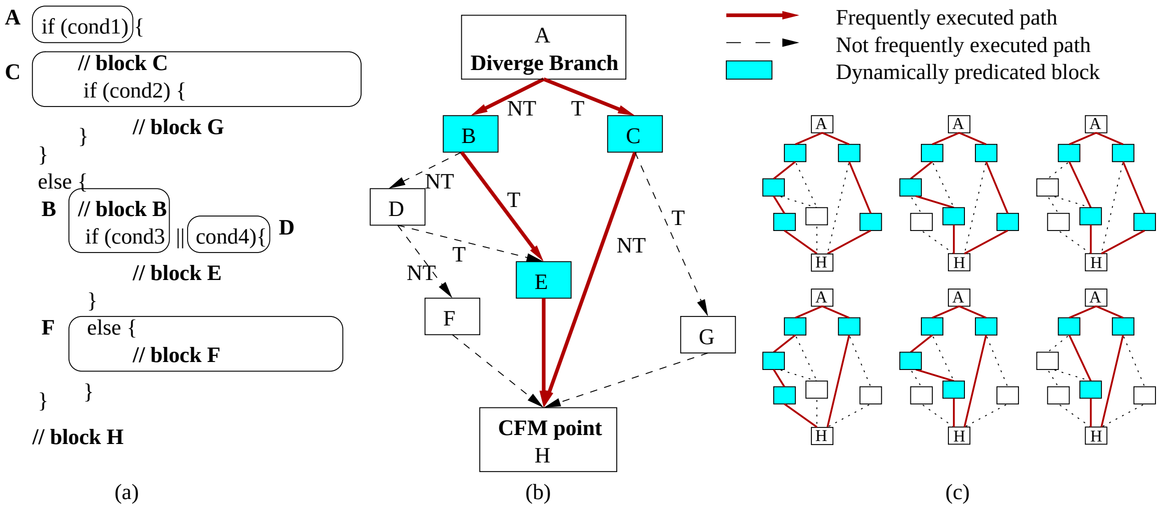
Chapter 1 described the three major problems/limitations of predicated execution: ISA support, the lack of adaptivity, and complex control-flow graphs. Wish branches in Chapter 4 was proposed to solve the second problem, the lack of adaptivity problem. However, wish branches inherit the limitations of software predication (ISA support and complex control flow graphs problem) with the exception that they can be applied to loop branches.

The goal of this chapter 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.

This chapter presents 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.

5.2 The Diverge-Merge Concept and Comparisons with Previous Work

5.2.1 Diverge-Merge Concept



```
A  if (cond1){
C      // block C
       if (cond2) {
              // block G
       }
   }
   else {
B      // block B
       if (cond3 || cond4){  D
              // block E
       }
F      else {
              // block F
       }
   }
   // block H
```

Figure 5.1: Control-flow graph (CFG) example: (a) source code (b) CFG (c) possible paths (hammocks) that can be predicated by DMP

Figure 5.1 shows a control-flow graph 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 [57].[1] In contrast, DMP considers frequently executed paths at run-time, so

[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 [5, 71]. Therefore, the compiler (e.g. ORC [57]) usually predicates all control-flow dependent basic blocks inside a region (the region is A,B,C,D,E,F,G and

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.

The compiler could predicate only blocks B, C, and E based on profiling [51] 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 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 5.1c 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 [43] 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

H in this example.). This problem can be mitigated with reverse if-conversion [81, 6] or by incorporating predicate information into the branch history register [5].

hammock when only frequently executed paths are considered.

5.2.2 The Basic DMP Operation

The compiler identifies conditional branches with control flow suitable for dynamic predication as *diverge branches*. A diverge branch is a branch instruction after which 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 5.3.11.

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 5.3.

5.2.2.1 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.

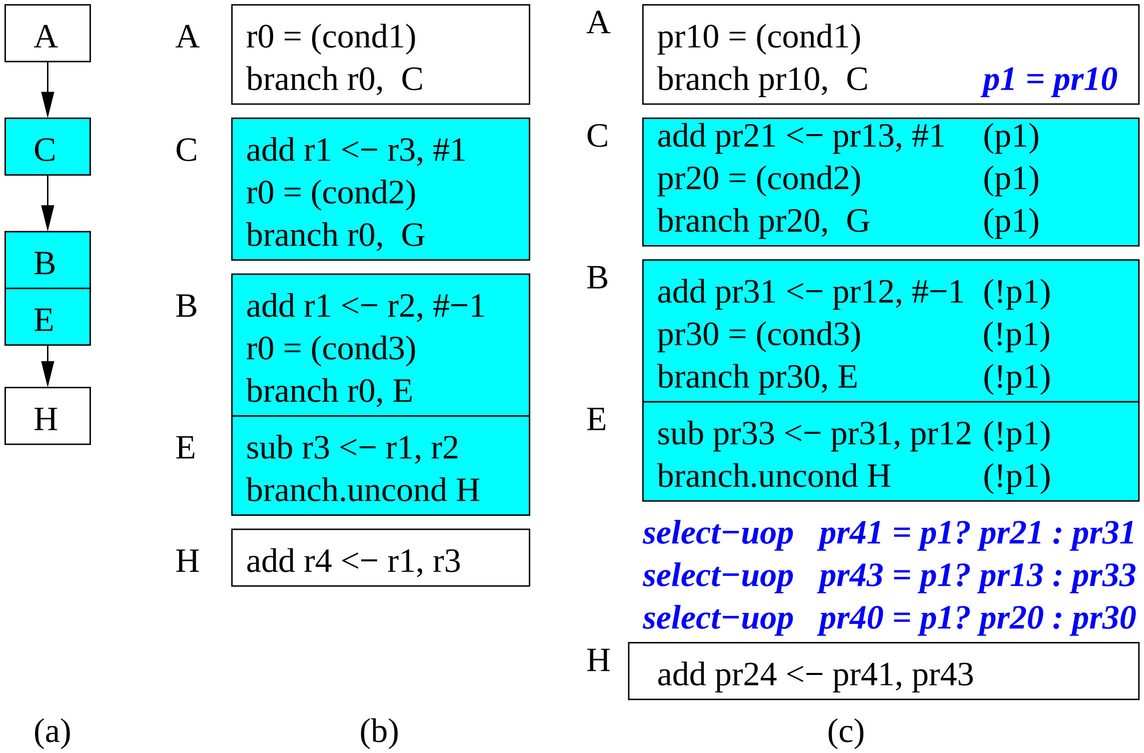
5.2.2.2 Select-μops

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 [24] 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

[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.

[3] Select-μops handle the merging of only register values. We explain how memory values are handled in Section 5.3.8.

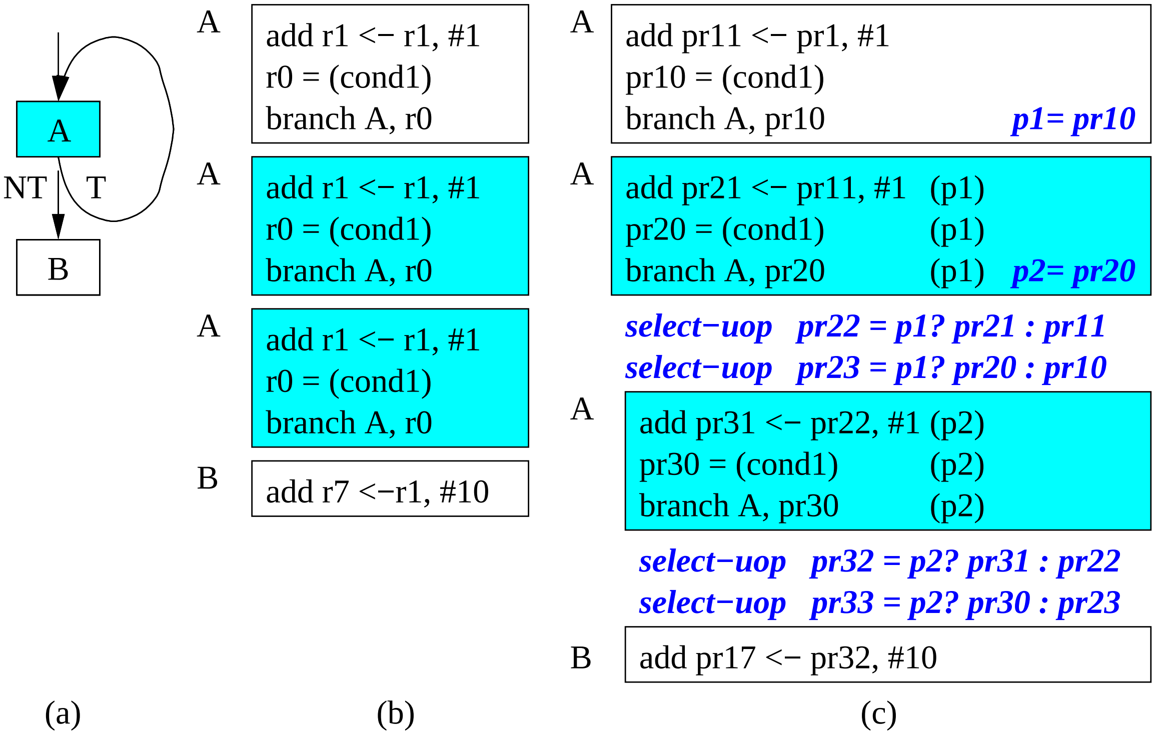
the CFM point is dependent on a register produced on either side of the hammock, it sources (i.e., depends on) the output of a select-μop. Such an instruction will be executed after the diverge branch is resolved. However, instructions that are not dependent on select-μops are executed as soon as their sources are ready without waiting for the resolution of the diverge branch. Figure 5.2 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.



```
A   r0 = (cond1)
    branch r0,  C

C   add r1 <− r3, #1
    r0 = (cond2)
    branch r0,  G

B   add r1 <− r2, #−1
    r0 = (cond3)
    branch r0, E

E   sub r3 <− r1, r2
    branch.uncond H

H   add r4 <− r1, r3
```

```
A   pr10 = (cond1)
    branch pr10,  C               p1 = pr10

C   add pr21 <− pr13, #1     (p1)
    pr20 = (cond2)           (p1)
    branch pr20,  G          (p1)

B   add pr31 <− pr12, #−1    (!p1)
    pr30 = (cond3)           (!p1)
    branch pr30, E           (!p1)

E   sub pr33 <− pr31, pr12   (!p1)
    branch.uncond H          (!p1)

    select−uop   pr41 = p1? pr21 : pr31
    select−uop   pr43 = p1? pr13 : pr33
    select−uop   pr40 = p1? pr20 : pr30

H   add pr24 <− pr41, pr43
```

Figure 5.2: An example of how the instruction stream in Figure 5.1b is dynamically predicated: (a) fetched blocks (b) fetched assembly instructions (c) instructions after register renaming



```
A   add r1 <− r1, #1
    r0 = (cond1)
    branch A, r0

A   add r1 <− r1, #1
    r0 = (cond1)
    branch A, r0

A   add r1 <− r1, #1
    r0 = (cond1)
    branch A, r0

B   add r7 <−r1, #10
```

```
A   add pr11 <− pr1, #1
    pr10 = (cond1)
    branch A, pr10                p1= pr10

A   add pr21 <− pr11, #1  (p1)
    pr20 = (cond1)        (p1)
    branch A, pr20        (p1)    p2= pr20

    select−uop   pr22 = p1? pr21 : pr11
    select−uop   pr23 = p1? pr20 : pr10

A   add pr31 <− pr22, #1 (p2)
    pr30 = (cond1)        (p2)
    branch A, pr30        (p2)

    select−uop   pr32 = p2? pr31 : pr22
    select−uop   pr33 = p2? pr30 : pr23

B   add pr17 <− pr32, #10
```

Figure 5.3: An example of how a loop-type diverge branch is dynamically predicated: (a) CFG (b) fetched assembly instructions (c) instructions after register renaming

5.2.2.3 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* in Chapter 4. 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 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 5.3 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.

5.2.3 DMP vs. Other Branch Processing Paradigms

We compare DMP with five previously proposed mechanisms in predication and multipath execution paradigms: dynamic-hammock-predication [43], software predication [3, 58], wish branches, selective/limited dual-path execution (dual-path) [32, 27], and multipath/PolyPath execution (multipath) [63, 45]. First, we classify control-flow graphs (CFGs) into five different categories to illustrate the differences between these mechanisms more clearly.

Figure 5.4 shows examples of the five different CFG types. Simple hammock (Figure 5.4a) is an `if` or `if-else` structure that does not have any nested branches inside the hammock. Nested hammock (Figure 5.4b) is an `if-else` structure that has multiple levels of nested branches. Frequently-hammock (Figure 5.4c) is a CFG that becomes a simple hammock if we consider only frequently executed paths. Loop (Figure 5.4d) is a cyclic CFG (`for`, `do-while`, or `while` structure). Non-merging control-flow (Figure 5.4e) is a CFG that does not have a control-flow merge point even if we consider only frequently executed paths.[4] Figure 5.5 shows the frequency of branch mispredictions due to each CFG type. Table 5.1 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.

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

[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.

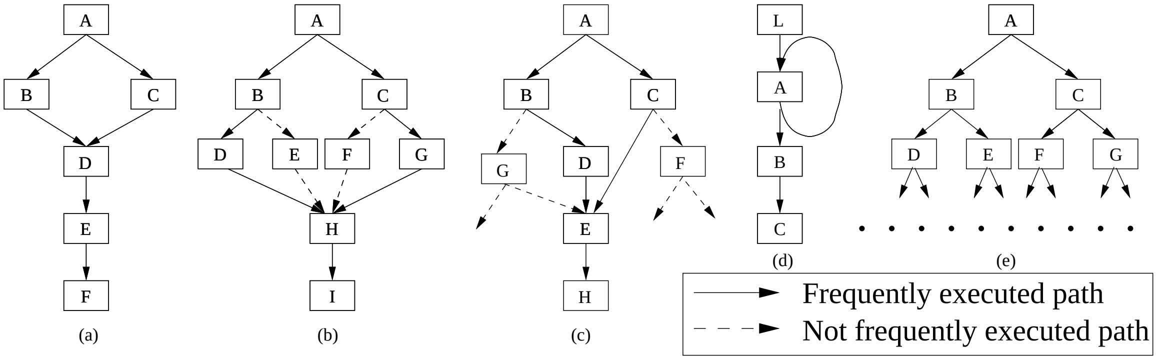


Figure 5.4: Control-flow graphs: (a) simple hammock (b) nested hammock (c) frequently-hammock (d) loop (e) non-merging control flow

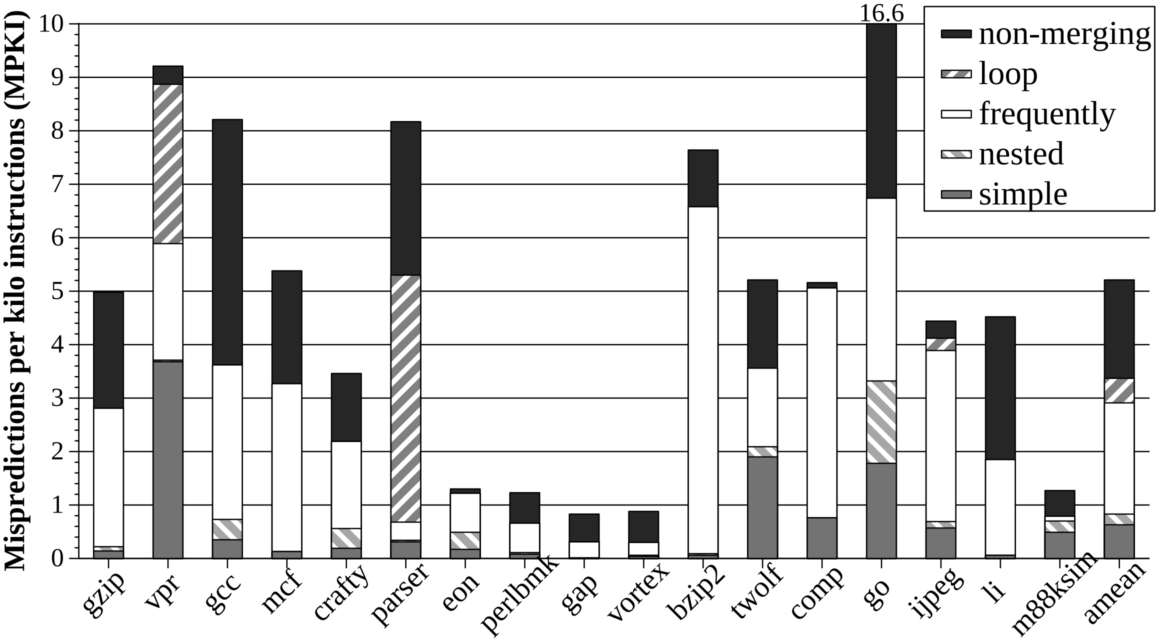


Figure 5.5: Distribution of mispredicted branches based on CFG type

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

Table 5.1: 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 5.4d) 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 ...

example, in the nested hammock case (Figure 5.4b), 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 [3, 58, 78, 50, 16, 57]. Note that hyperblock formation [51] 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 even predicate 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., the branch at B is control-dependent on the branch at A in Figure 5.4b), 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 in only 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 5.4a), 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 5.5.

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 5.1 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 5.1. Hence, we expect DMP to eliminate branch mispredictions more efficiently (i.e., with less overhead) than the other processing paradigms.

5.3 Implementation of DMP

5.3.1 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.[5] 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. The BTB is extended to store diverge branch type information and CFM information.
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

[5] The compiler could also provide a hint bit to indicate that it is better to enter dpred-mode regardless of the confidence estimation. This additional mechanism is called *short-hammocks* and it will be explained more in Chapter 6

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.

5.3.2 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.

5.3.3 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 uses 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 5.4c, 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. **Non-preemptive 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.

Figure 5.6 shows an example of non-preemptive policy. Assume that the processor enters dpred-mode when it fetches diverge branch A. Later, the processor fetches diverge branch D that also has low-confidence. At that moment, the processor has already fetched blocks B, C, and G. With non-preemptive policy, the processor exits dpred-mode for branch A and re-enters dpred-mode for diverge branch D. The processor allocates a new predicate id register for diverge branch D and then predicates instructions with the new predicate id. Right before the processor enters dpred-mode for the diverge branch D, it creates a checkpoint for the register alias table, GHR, RAS, and PC address associated with path-C. The instructions on path-C that are older than the checkpoint will still be sent to the pipeline. When branch A is resolved and if path-C is the correct path, all the instructions on path-C will have TRUE predicate values and all the instructions on path-B will have FALSE predicate values. Similar to the loop mechanism that will be described in Section 5.3.6, predicate ids that are generated later than the predicate id for diverge branch C will broadcast FALSE values. The processor restores the checkpoint associated with path-C and then restarts fetch. If path-B is correct and path-C is wrong, all the instructions on path-C would have FALSE predicate values.

Note that the storing/restoring of checkpoints on path C needs to happen even in the baseline processor to support recovery from branch mispredictions. The processor has to create a checkpoint regardless of whether it enters dpred-mode again or not. Hence, non-preemptive policy does not add significant extra hardware overhead to the pipeline.

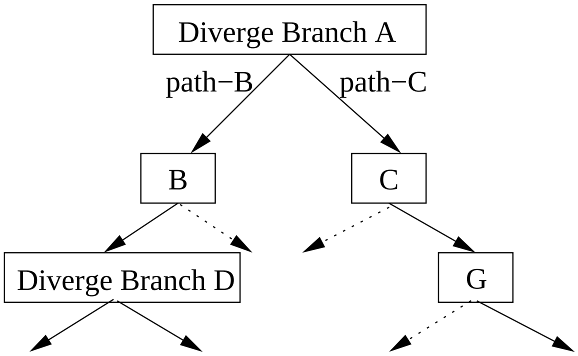


Figure 5.6: Control-flow graph (CFG) example for non-preemptive policy

5.3.4 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 [79]. 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 as follows:

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.[6] 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

[6] 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.

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.

5.3.5 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[7], 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 5.3.6. 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.

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

5.3.6 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.
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.[8] 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.

5.3.7 Instruction Execution and Retirement

Dynamically predicated instructions are executed just like other instructions (except for store-load forwarding described in Section 5.3.8). Since these instructions depend on

[8] 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).

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.[9] The predicate register associated with dpred-mode is released when the last predicated instruction is retired.

5.3.8 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. As one would expect, 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.,

[9] In a high performance 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.

on the same dynamically predicated path). If forwarding is not possible, the load waits. Note that this mechanism and the structures to support it are the same as the store-load forwarding mechanism in dynamic-hammock-predication [43]. 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.

5.3.9 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.

5.3.10 Hardware Complexity Analysis

DMP increases hardware complexity compared to current processors but is an energy efficient design as we will show in Section 5.5.5. Some of the hardware required for DMP already exists 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 5.2 summarizes the additional hardware 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 5.2: 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

5.3.11 ISA Support for Diverge Branches

This section presents 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 a maximum of three CFM points per diverge branch. To support early exit (Section 5.3.3), 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).

5.4 Methodology

5.4.1 Simulation Methodology

The evaluation is done with an execution-driven simulator [11] 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 5.3. A less aggressive processor is also modeled to evaluate the DMP concept in a configuration similar to today's processors. Table 5.4 shows the parameters of the less aggressive processor that are different from the baseline processor.

Table 5.3: 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 [36] 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 [76, 8]
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) [77]
DMP Support	2KB (12-bit history, threshold 14) enhanced JRS confidence estimator [35, 30] 32 predicate registers; 3 CFM registers (also see Table 5.2)

The experiments are run using the 12 SPEC CPU 2000 integer benchmarks and five of the eight SPEC 95 integer benchmarks.[10] Table 5.5 shows the characteristics of the

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

Table 5.4: Less aggressive baseline processor 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 [36]; 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

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 [46] to reduce simulation time. In all the IPC (retired Instructions Per Cycle) performance results shown in the rest of the dissertation for DMP, instructions whose predicate values are FALSE and select-μops inserted to support dynamic predication do not contribute to the instruction count.

5.4.2 Modeling of Other Branch Processing Paradigms

5.4.2.1 Dynamic-Hammock-Predication

Klauser et al. [43] 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.

5.4.2.2 Dual-path

Several design choices for dual-path processors were proposed [32, 27, 45, 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

Table 5.5: Characteristics of the benchmarks: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), IPC, potential IPC improvement with perfect branch prediction (PBP IPC Δ) in both baseline processor and less aggressive processor.

	Insts	Diverge br.	All br	Code size Δ	baseline processor		less aggressive processor	
					IPC	PBP IPC Δ	IPC	PBP IPC Δ
gzip	249M	84	1.6K	0.12%	2.02	90%	1.77	39%
vpr	76M	434	4.2K	0.35%	1.50	229%	1.39	84%
gcc	83M	1245	29.5K	0.23%	1.25	96%	0.98	46%
mcf	111M	62	1.4K	0.1 %	0.45	113%	0.52	58%
crafty	190M	192	5.1K	0.13%	2.54	60%	1.76	27%
parser	255M	37	3.7K	0.03%	1.50	137%	1.36	65%
eon	129M	116	4.9K	0.01%	3.26	21%	2.05	9%
perlbmk	99M	92	9.4K	0.03%	2.27	15%	1.36	7%
gap	404M	79	4.6K	0.03%	2.88	15%	2.03	9%
vortex	284M	250	13K	0.09%	3.37	16%	1.73	8%
bzip2	316M	74	1.4K	0.11%	1.48	94%	1.39	46%
twolf	101M	235	4.7K	0.16%	2.18	112%	1.71	46%
compress	150M	16	0.6K	0.02%	2.18	139%	1.79	50%
go	137M	117	7.7K	0.08%	0.97	227%	0.86	101%
ijpeg	346M	48	2K	0.04%	2.73	93%	2.05	37%
li	248M	18	1.2K	0.02%	2.15	60%	1.69	34%
m88ksim	145M	158	1.7K	0.13%	3.27	24%	2.10	12%

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.

5.4.2.3 Multipath

The modeled multipath processor starts fetching from both paths *every time* it encounters a low-confidence branch, similar to PolyPath [45]. 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.

5.4.2.4 Limited Software Predication

Since the Alpha ISA does not support full predication, we model limited software predication[11] 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 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 [79] (similarly to DMP) is employed so that predicated instructions can be executed before the predicate value is ready.

5.4.2.5 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.

[11] We call it limited software predication, because our software predication does not model the compiler optimization effect on if-conversion

5.4.3 Power Model

We incorporated the Wattch infrastructure [7] 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 [7]. 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.

5.4.4 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. A detailed evaluation will be presented in Chapter 6. We used the *train* input sets to collect profiling information.

5.5 Results

5.5.1 Performance of the Diverge-Merge Processor

Figure 5.7 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,[12] 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 5.5. 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 5.3.2) 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 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

[12] Klauser et al. [43] reported average 5% performance improvement for dynamic-hammock-predication, Farrens et al. [27] reported average 7% performance improvement for dual-path (with extra execution resources to support dual-path), and Klauser and Grunwald [44] 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.

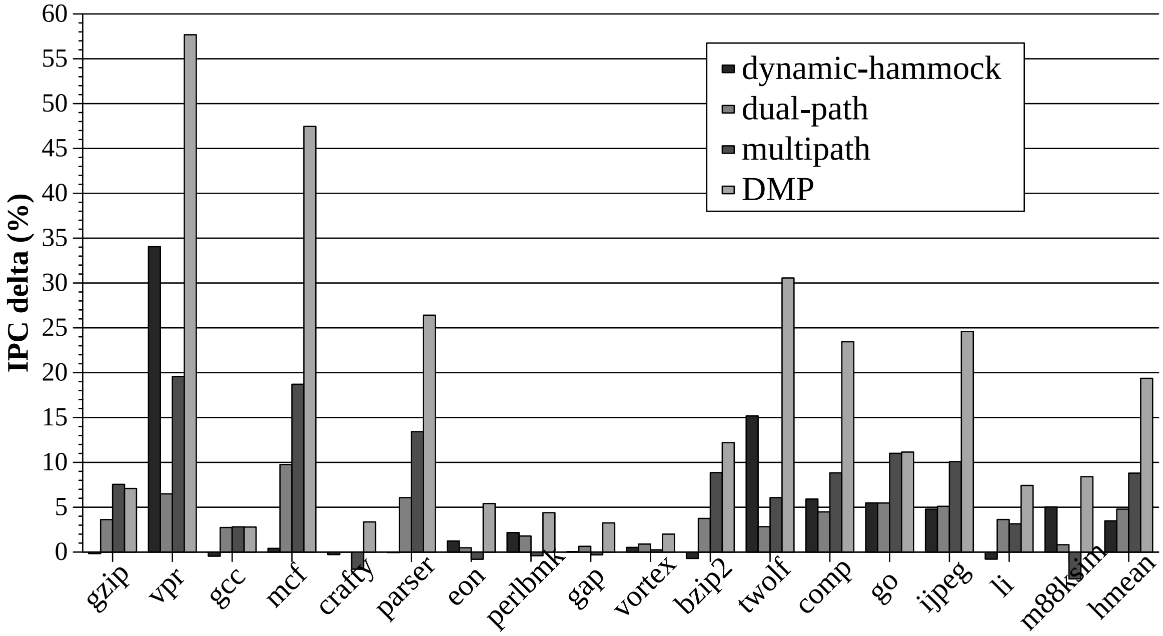


Figure 5.7: Performance improvement provided by DMP vs. dynamic-hammock-predication, dual-path, and multipath execution

in Table 5.1. Figure 5.8 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 5.9 shows, dual-path reduces pipeline flushes by 18% whereas DMP reduces them by 38%.

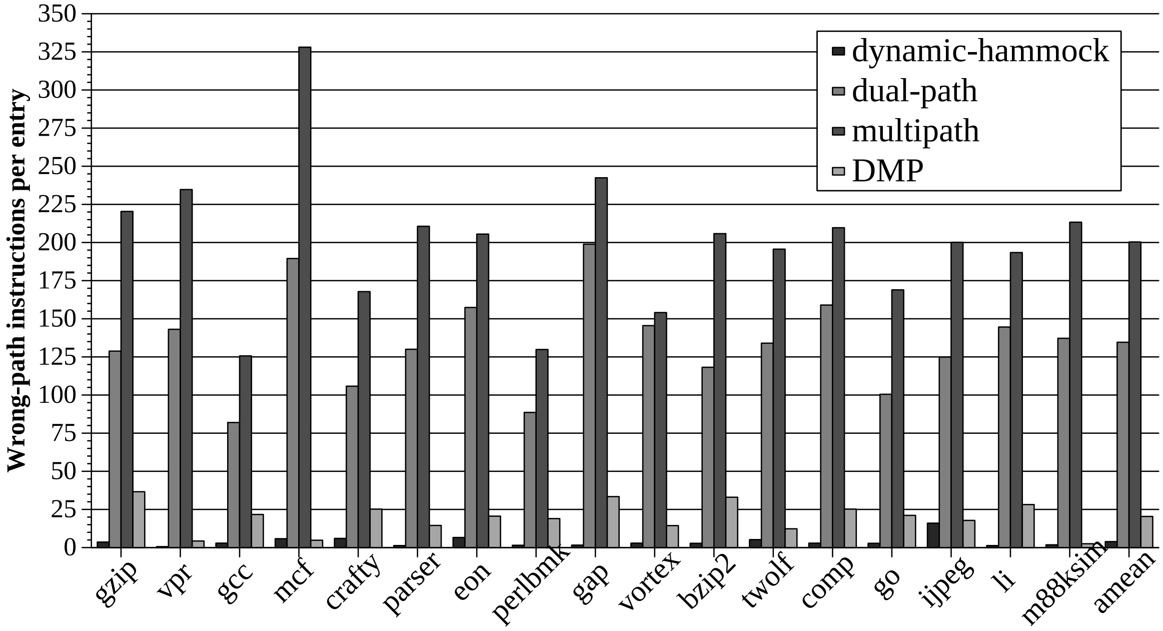


Figure 5.8: Fetched wrong-path instructions per entry into dynamic-predication/dual-path mode (i.e., per low-confidence branch)

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 5.8), its average performance improvement is much less than that of DMP.

5.5.2 Comparisons with Software Predication and Wish Branches

Figure 5.10 shows the execution time reduction over the baseline for limited software predication and wish branches. Since the number of executed instructions is different

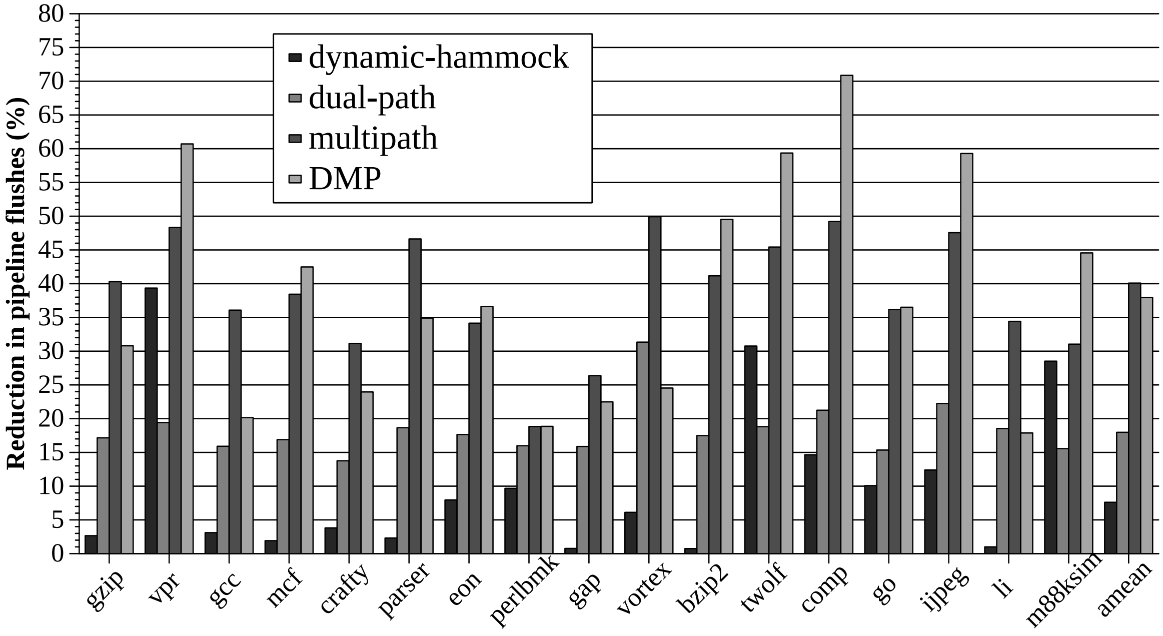


Figure 5.9: % reduction in pipeline flushes

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.

There are some differences between results of Chapter 4 and the results of Chapter 5 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

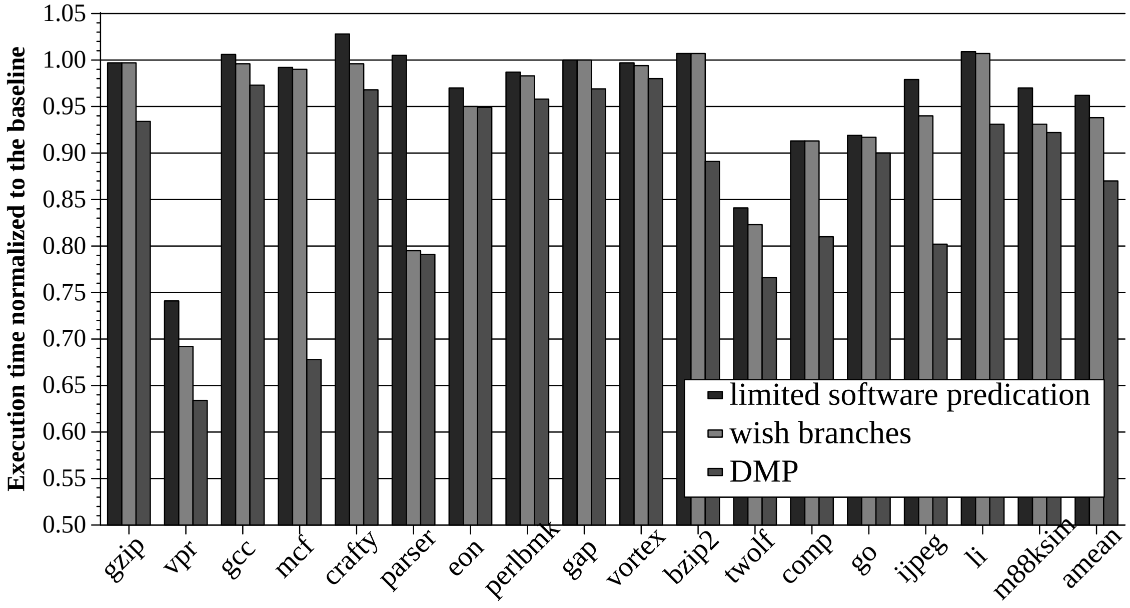


Figure 5.10: DMP vs. limited software predication and wish branches

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 5.5. Only parser does not have many frequently-hammocks, so wish branches and DMP perform similarly for this benchmark. Figure 5.11 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.

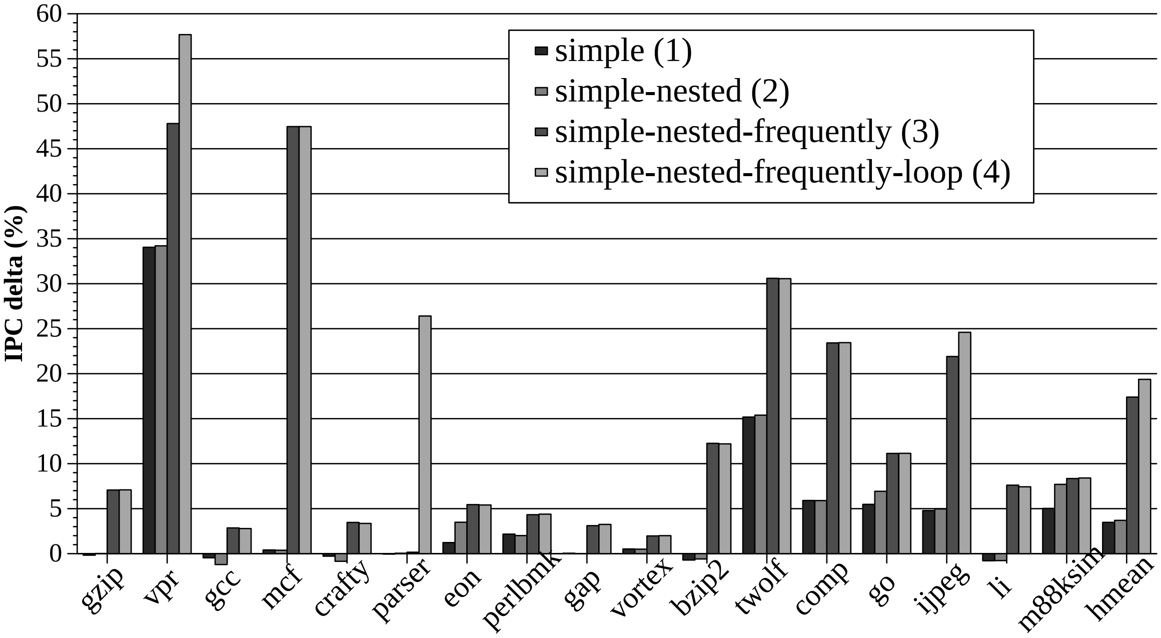


Figure 5.11: DMP performance when different CFG types are dynamically predicated

5.5.3 Analysis of the Performance Impact of Enhanced DMP Mechanisms

Figure 5.12 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 5.3.2. 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 5.3.3). 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-nonpre* also adopts the *non-preemptive Policy* (Section 5.3.3) to exit dpred-mode early, increasing the performance benefit of DMP to 19.3%. Non-preemptive 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.

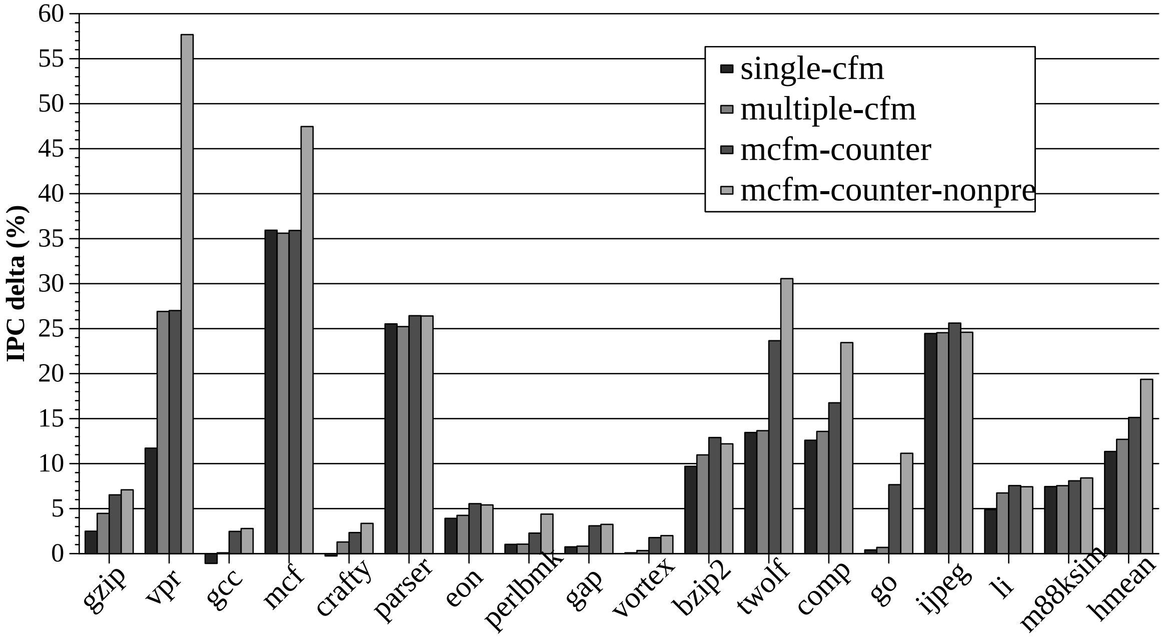


Figure 5.12: Performance impact of enhanced DMP mechanisms

5.5.4 Sensitivity to Microarchitecture Parameters

5.5.4.1 Evaluation on the Less Aggressive Processor

Figure 5.13 shows the performance benefit for dynamic-hammock-predication, dual-path, multipath, and DMP on the less aggressive baseline processor and Figure 5.14 shows

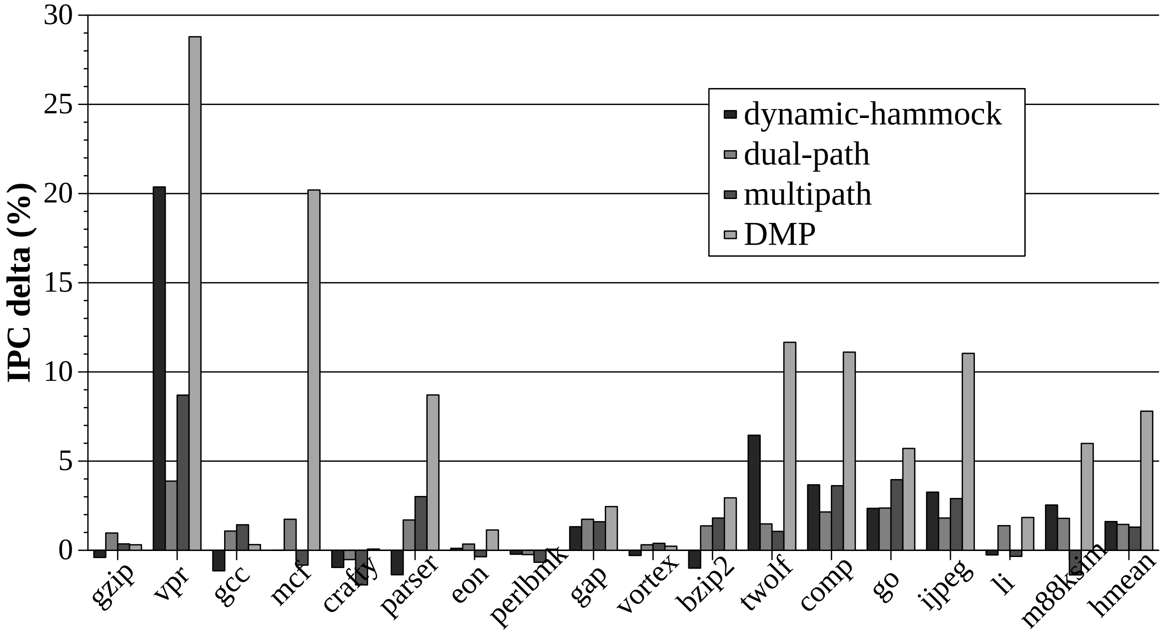


Figure 5.13: Performance comparison of DMP versus other paradigms (hardware oriented) on the less aggressive processor

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%.

5.5.4.2 Effect of a Different Branch Predictor

We also evaluate DMP with a recently developed branch predictor, O-GEHL [68]. 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 5.15

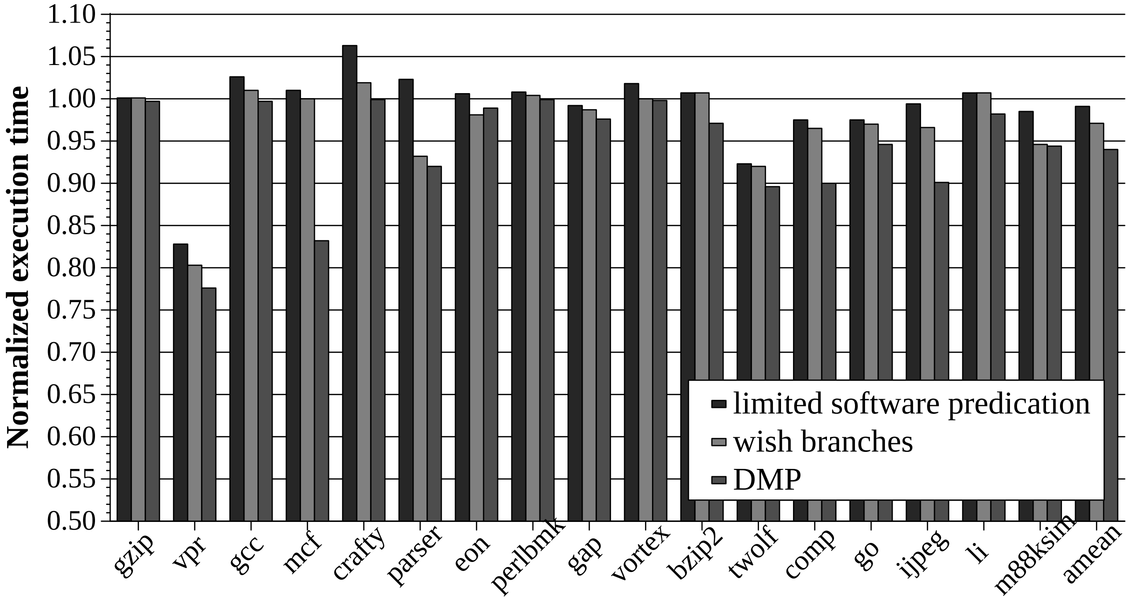


Figure 5.14: Performance comparison of DMP versus other paradigms (compiler oriented) on the less aggressive processor

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 5.16 shows the effect of replacing the baseline processor's perceptron predictor with a less complex 16KB gshare branch predictor (gshare-16KB). DMP with a 16KB gshare branch predictor provides 20.3% performance improvement, which is slightly better than the 19.3% with a perceptron branch predictor (perceptron-64KB). Even if DMP employs a larger gshare branch predictor (32KB, 64KB), the performance benefit of DMP is

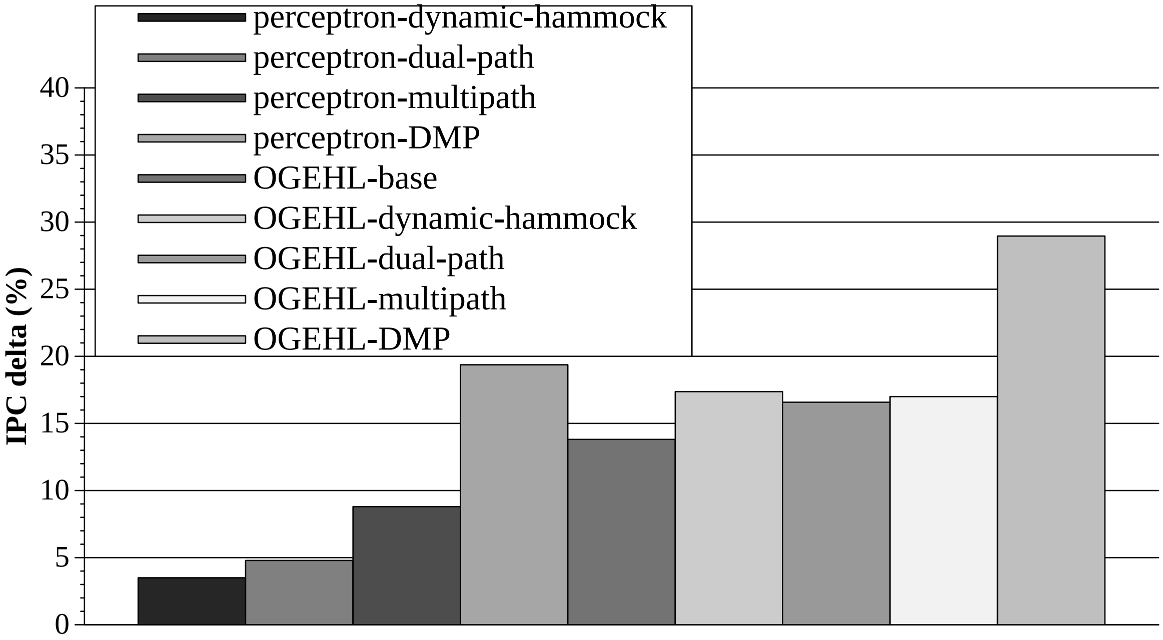


Figure 5.15: DMP performance with different branch predictors

not reduced even though actual IPC performance is improved. The results show that DMP is effective at reducing the branch misprediction penalty with other branch predictors.

5.5.4.3 Effect of Confidence Estimator

Figure 5.17 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 [35], which has 14% PVN ($\simeq$ accuracy) and 70% SPEC ($\simeq$ coverage) [30].[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-

[13] These numbers are actually lower than what was previously published [30] because our baseline branch predictor uses a different algorithm and has a much higher prediction accuracy than that of [30].

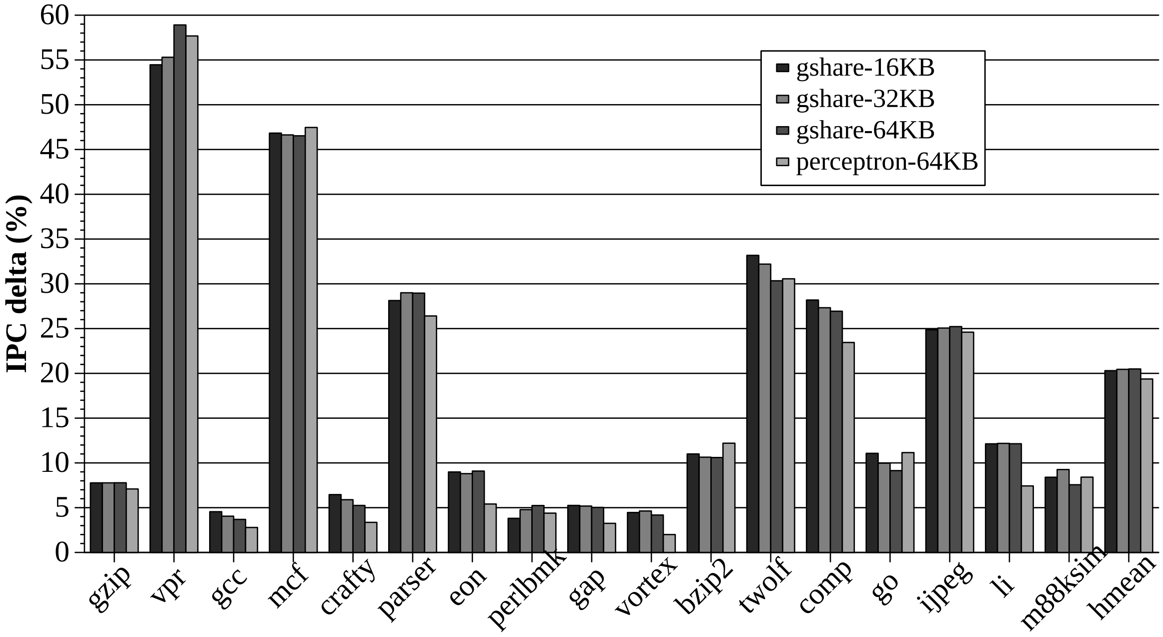


Figure 5.16: DMP performance with gshare branch predictors

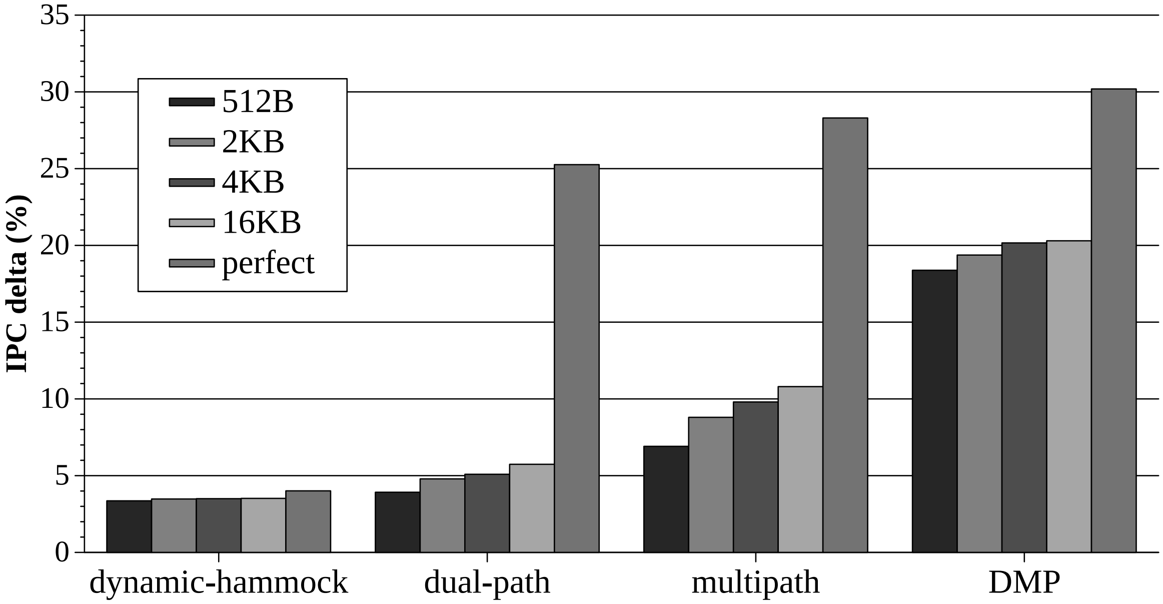


Figure 5.17: Effect of confidence estimator size on performance

path/multipath has very high overhead as shown in Figure 5.8, 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 5.5.1), (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 5.18 shows the accuracy and the coverage of the confidence estimator along with the performance improvement of DMP when we vary the threshold (N) of the confidence estimator. When N increases, branches are more likely to be estimated as low confidence [35], so the accuracy decreases but the coverage increases. Since the accuracy does not drop as fast as the coverage improves, DMP results in the best performance improvement when the coverage of the confidence estimator is the highest.

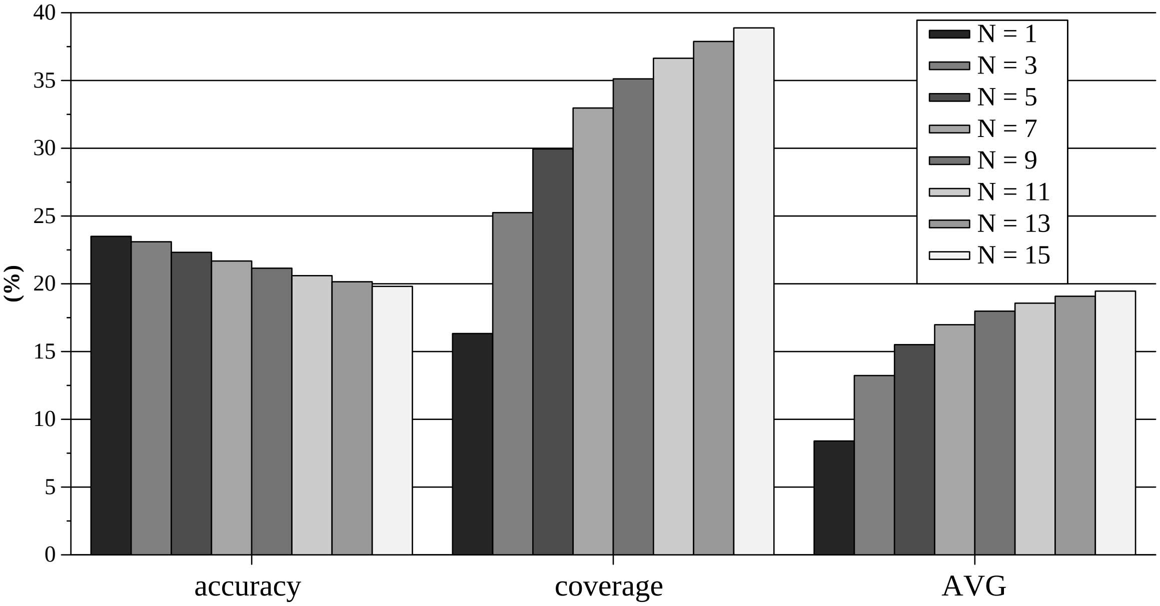


Figure 5.18: Confidence estimator thresholds

Our evaluation employs a JRS confidence estimator. Figure 5.19 shows the performance improvement with a perceptron confidence estimator [2]. The results show that the perceptron based confidence estimator provides 9.11% performance improvement, which is much less than when the DMP employs the JRS confidence estimator. The main reason is that the perceptron confidence estimator has lower accuracy and lower coverage than the JRS confidence estimator. Our experiments show that the perceptron predictor has 5-10% accuracy with 20-30% coverage when used with our baseline's branch predictor.

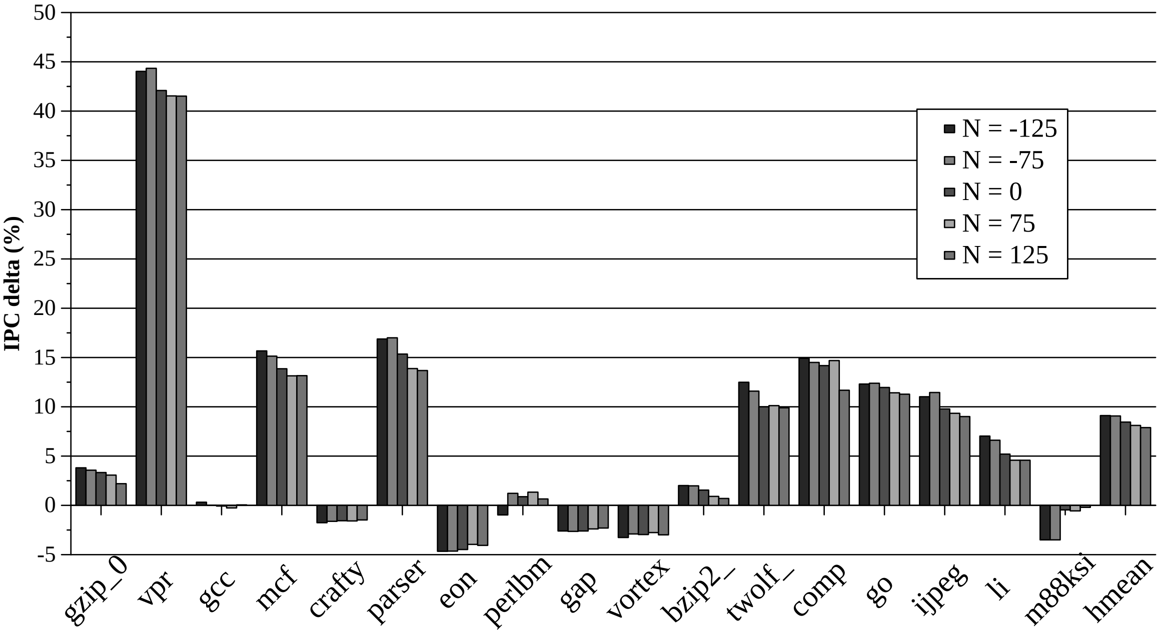


Figure 5.19: DMP with a perceptron based confidence estimator

5.5.5 Power Analysis

Figure 5.20 (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.

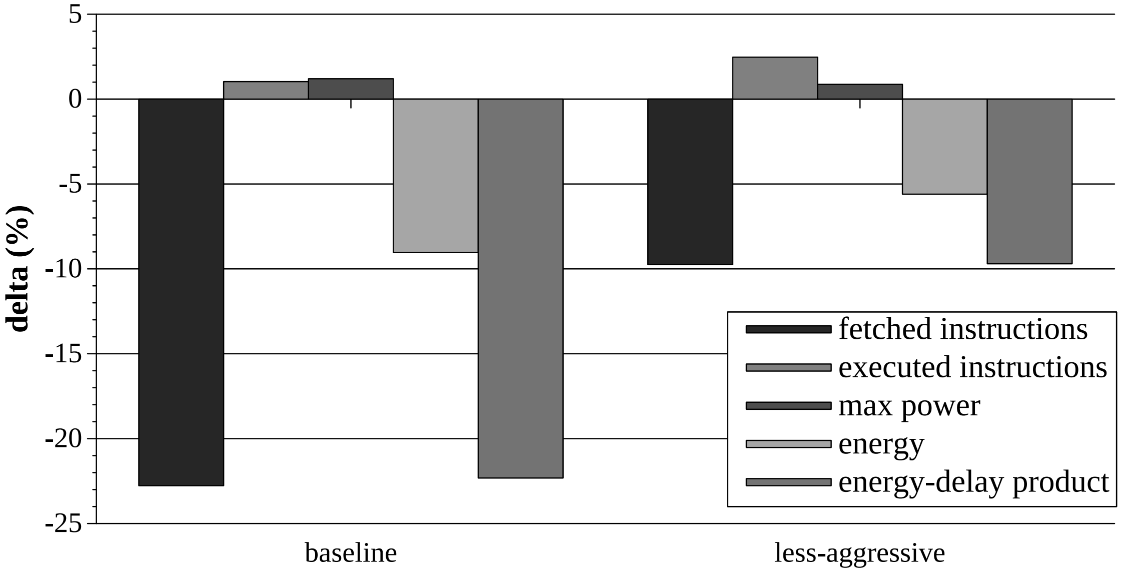


Figure 5.20: Power consumption comparison of DMP with baseline processor (left) and less aggressive baseline processor (right)

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 by reducing pipeline flushes due to branch mispredictions. DMP is an energy-efficient design even in the less aggressive processor configuration as Figure 5.20 (right) shows.

Table 5.7 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

Table 5.6: Power and energy comparison of different branch processing paradigms

	Baseline processor					
	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%
Energy Δ	-9.0%	-0.7%	-2.2%	4.7%	-1.5%	-2.9%
Energy $\times$ Delay Δ	-22.3%	-0.9%	-7.0%	-4.3%	-1.8%	-6.1%

Table 5.7: Power and energy comparison of different branch processing paradigms in less aggressive baseline processor

	Less aggressive baseline processor					
	DMP	dyn-ham.	dual-path	multipath	SW-pred	wish br.
Max power Δ	0.9%	0.8%	0.8%	4.3%	0.1%	0.4%
Energy Δ	-5.6%	-0.8%	1.1%	3.7%	-0.1%	-1.5%
Energy $\times$ Delay Δ	-9.7%	-0.5%	0.5%	2.2%	1.2%	-2.1%

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.

5.5.6 The Diverge-Merge Processor Design Configuration

5.5.6.1 Select-μop vs. Conditional Expression Mechanism

DMP uses the select-μop mechanism to effectively execute dynamic predicated instructions. Predicated instructions (instructions fetched during dpred-mode) can be executed before predicate value is known. Dynamic predication can also be implemented using C-style conditional expressions [74]. For example, `(p1)r1=r2+r3` instruction is converted to the μ`op r1 = (p1) ?  (r2+r3) :  r1`, which is similar to a select-μop. If the predicate is TRUE, the instruction performs the computation and stores the result into the destination register. If the predicate is FALSE, the instruction simply moves the old value of the destination register into its destination register. With the C-style conditional expression mechanism, the processor can reduce the overhead of select-μops (generation of select-μops and execution of select-μops). However, not all instructions fetched during dpred-mode can be executed until the diverge branch is resolved. The processor also needs to support one more register read port to support the reading of the old value from the destination register.

Figure 5.21 shows the performance difference between conditional expression mechanism and select-μop mechanism. Since the execution delay due to predicated instructions is not high in most benchmarks, both mechanisms show similar performance benefit except in mcf. Mcf is a memory-limited benchmark. In mcf, critical instructions that generate L2 cache miss requests cannot be executed until the predicate value is ready if the processor employs the conditional expression mechanism. This results in significant delay in handling L2 misses, which reduces memory level parallelism, thereby leading to the significant performance degradation. These results show that if a processor cannot afford a select-μop generation mechanism, converting predicated instructions to conditional expressions could be useful. However, in that case, the compiler needs to be aware of long latency operations when generating code for the DMP processor. (In other words, if the instructions between a diverge branch and the CFM point are more likely to generate cache misses, it is better not to mark the branch as a diverge branch. Note that this problem also exists in generating predicated code for an out-of-order processor.)

5.5.6.2 Fetch Mechanisms

DMP fetches instructions from both paths in a round-robin manner during dynamic predication mode. An alternative design option is to fetch one-path first until the CFM point and the other path next instead of fetching from two paths in alternate cycles. This mechanism is called *fetch-one-by-one*. The benefit of fetch-one-by-one is that the processor does not require two active register alias tables and two return address stacks. In

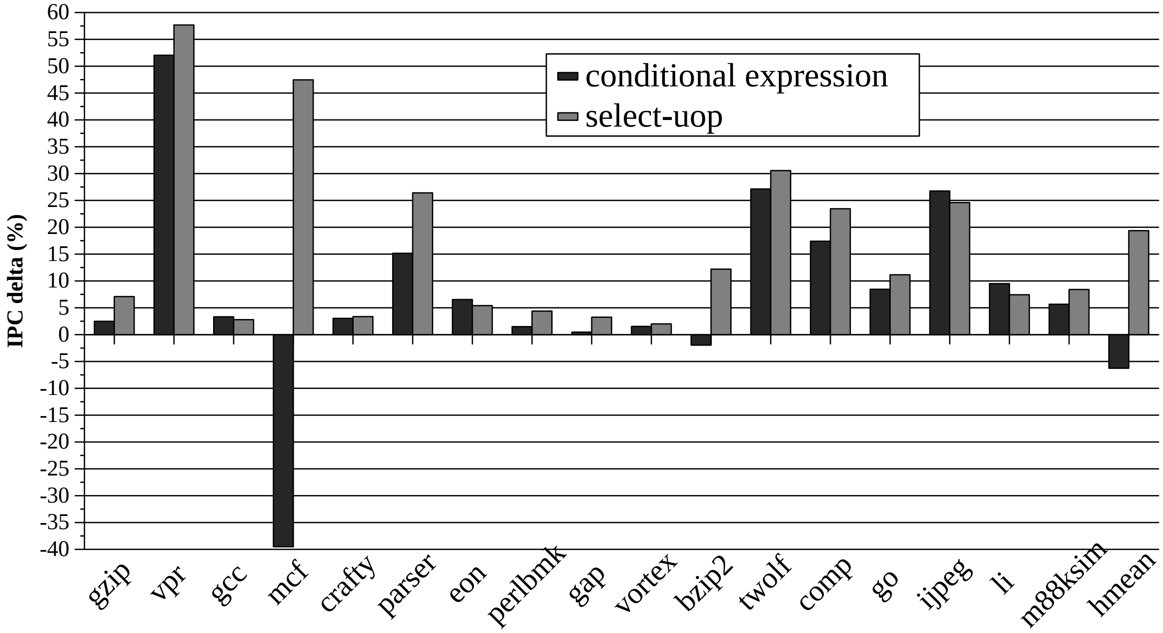


Figure 5.21: Select-μop vs. conditional expression

fetch-one-by-one, when the processor enters dynamic predication mode, it creates a checkpoint of the register alias table and fetches the predicted path first (this is the path predicted to be followed by the branch predictor). When the processor reaches the CFM point on the predicted path, it restores the checkpoint and starts fetching from the other path. It fetches from the other path until it reaches the CFM point again. After that, the processor inserts select-μops by comparing two register alias tables (one is the active register alias table and the other is stored in the checkpoint). Figure 5.22 shows the performance benefit comparison of fetch-one-by-one and round-robin schemes. Fetch-one-by-one performs slightly worse than the round-robin scheme. However, fetch-one-by-one still provides 17.7% performance improvement. Hence, fetch-one-by-one could be a viable design option if design constraints prohibit the maintenance of two active register alias tables and return address stacks at the same time.

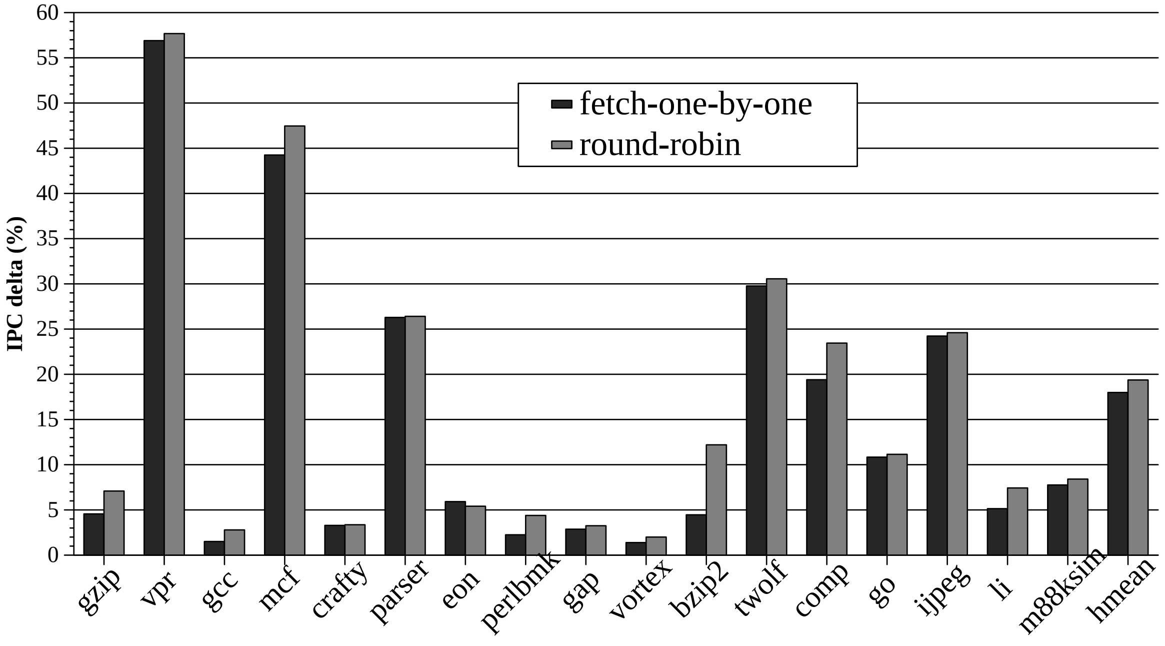


Figure 5.22: Different fetch mechanisms

5.5.7 DMP Analysis

Table 5.8 summarizes the important dynamic predication mode statistics. *Useful dpred-mode* shows how often dpred-mode is useful, i.e., the fraction of dpred-mode instances initiated by an actually-mispredicted branch. The results show that about 20% of the dpred-mode instances are useful. The usefulness of a dpred-mode instance is strongly dependent on the accuracy of the confidence estimator. As Figure 5.18 shows, the accuracy of the confidence estimator is about 20%. This explains why useful dpred-mode is approximately 20%.

Merge probability shows how often both paths merge at the same CFM point. On average, 58.7% dpred-mode instances result in both paths merging at the same CFM point. However, benchmark that see significant performance benefits –such as vpr, mcf, twolf, li, and ijpeg– have more than 70% merge probability. Hence, it is important for the compiler to choose candidate "hammocks" that are likely to merge.

average select-μops shows the average number of select-μops generated for each dynamic dpred-mode. On average about 7 select-μops are generated. Hence, the overhead of select-μops is not significant compared to the instructions saved due to an eliminated pipeline flush.

Table 5.8: Characteristics of dpred-mode

	useful dpred-mode	merge probability	average select-μops
gzip	20.1 %	37.8 %	14.33
vpr	22.6 %	71.7 %	5.17
gcc	17.7 %	40.3 %	7.00
mcf	17.7 %	78.9 %	3.22
crafty	13.4 %	49.9 %	7.67
parser	18.3 %	50.3 %	5.78
eon	15 %	50.2 %	3.69
perlbmk	20.2 %	60.9 %	7.17
gap	23.4 %	48.7 %	6.49
vortex	15.3 %	59.2 %	5.43
bzip2	21.2 %	19.6 %	13.98
twolf	20.2 %	77.5 %	4.86
compress	21.5 %	68.5 %	10.24
go	19.6 %	42.7 %	9.13
ijpeg	34.3 %	81.8 %	6.70
li	26.3 %	70.7 %	8.65
m88ksim	15.6 %	90.1 %	2.22
amean	20.2 %	58.7 %	7.16

5.5.8 Diverge-Merge Processor and Pipeline Gating

Pipeline gating mechanism was proposed to save energy by reducing the speculative instruction fetch [52]. The processor stops the front-end if there is a certain number of low-confidence branch instructions inside the pipeline. This pipeline gating mechanism can be applied to the diverge-merge processor also. If there is a certain number of low-confidence non-diverge branches inside the pipeline, the processor gates the front-end just as in the original pipeline gating mechanism. Figure 5.23 shows the performance improvement and energy consumption of pipeline gating on top of DMP. *pg-th* is the number of low-confidence branches in the pipeline, which triggers the processor gates the pipeline. As pg-th increases, the energy consumption reduction becomes closer to that of DMP without pipeline gating. pg-th 15 shows the best Energy-Delay-Product reduction.

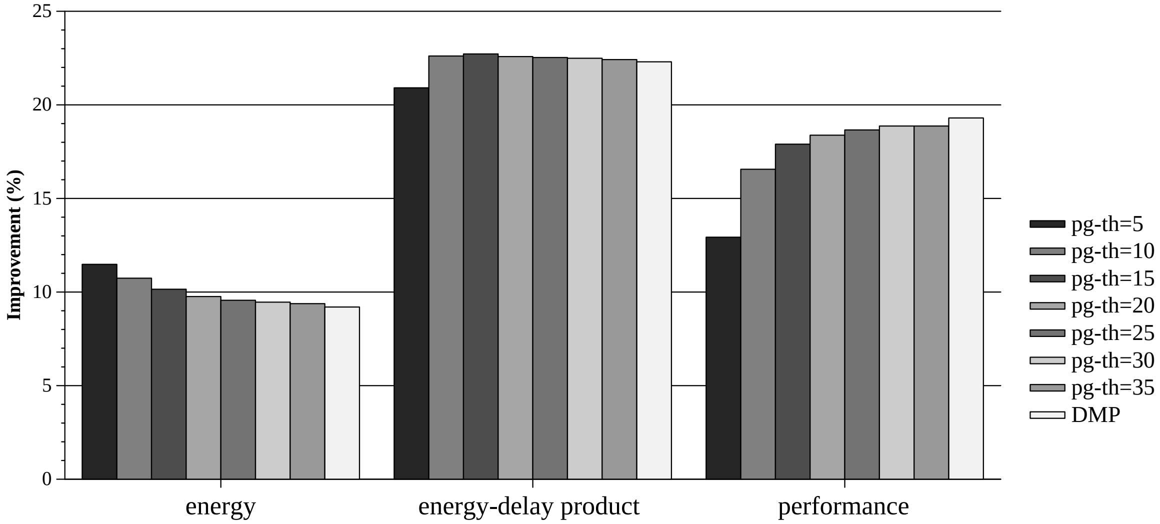


Figure 5.23: Pipeline gating mechanisms on DMP

These results show that pipeline gating reduces energy consumption at the cost of moderate performance loss whereas DMP can reduce energy consumption while at the same time increasing performance. However, both mechanisms can be used together to achieve the best energy-delay product.

5.6 Summary

This chapter 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 5.1).
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 5.1 and Figure 5.8).

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.

Furthermore, diverge-merge processor increases the applicability of predication in 4 major ways:

1. It significantly reduces the ISA support required for predicated execution by eliminating the need for ISA-visible predicate registers and predicated instructions. *This*

would enable more processors to support predicated execution regardless of the ISA they implement.

2. It makes the benefits of predicated execution applicable to a large set of CFGs, especially complex code structures, with less overhead than static predication. *This enables complex applications with large, complicated CFGs to benefit from predicated execution.*
3. It makes predicated execution adaptive to run-time branch behavior, which eliminates the performance degradation sometimes caused by statically predicated code. Unlike static predication or hyperblocks, the decision of which paths are predicated and when they are predicated is not statically fixed by the compiler. *This enables more applications to benefit from predicated execution, especially those with branch phase behavior and whose run-time behavior differs significantly from profile-time behavior.*
4. It makes code generation for predicated execution simpler because the compiler does not need to decide, without run-time information, which branches *should be* if-converted. *This simplifies the compile-time profiling and cost-benefit analysis schemes required for predication.*

Chapter 6

Compiler Algorithms for the Diverge-Merge Processor Architecture

6.1 Introduction

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. A diverge branch can be part of either a simple hammock or a frequently-hammock. 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. This chapter describes the compiler and profiling algorithms for a DMP processor and explores the tradeoffs involved in the design of these algorithms. This chapter evaluates the impact of these algorithms on the performance of a DMP processor and provides insights into what is important to consider in the design of such algorithms.

6.2 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. 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 [58, 51, 78, 53]). In frequently-hammocks, the probability that both paths after a diverge branch reach a CFM point is another factor that determines whether or not dynamically predicating the diverge branch would provide benefit. 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 6.3.

6.2.1 Diverge Branch Candidates

There are four types of diverge branches based on the CFG types they belong to: simple hammock (Figure 5.4a), nested hammock (Figure 5.4b), frequently-hammock (Figure 5.4c), and loop (Figure 5.4d). The descriptions of each CFG types are explained in Section 5.2.3.

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.

6.2.2 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 [22] 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.

6.2.3 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.

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

```
for each conditional branch $B$ executed during profiling do
    Compute $IPOSDOM(B)$ of $B$
    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 $\geq MIN\_EXEC\_PROB$.
    for each basic block $X$ reached on both the taken and the not-taken directions of $B$ do
        $p_T(X) \leftarrow$ edge-profile-based probability of $X$ being reached on the taken direction of $B$
        $p_{NT}(X) \leftarrow$ edge-profile-based probability of $X$ being reached on the not-taken direction of $B$
        $probability\ of\ merging\ at\ X \leftarrow p_T(X) * p_{NT}(X)$
        if ($probability\ of\ merging\ at\ X \geq MIN\_MERGE\_PROB$) then
            add $X$ as a CFM point candidate for $B$
        end if
    end for
    select up to $MAX\_CFM$ CFM point candidates for $B$, the ones with the highest $probability\ of\ merging\ at\ X$
end for
```

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.[1] If the profiled probability of reaching a CFM point candidate is lower than a threshold (MIN_MERGE_PROB), the CFM point

[1] 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 [32].

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

6.2.3.1 A chain of CFM Points

Figure 6.1 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.[2]

[2] 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 6.1, $probability\ of\ merging\ at\ C$ = $p_T(C) * p_{NT}(C)$ = 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.

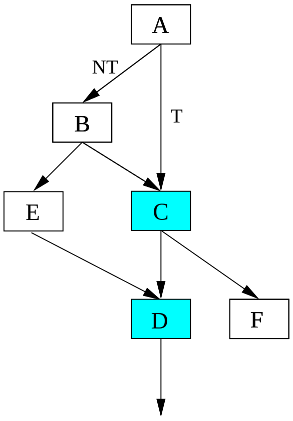


Figure 6.1: Example of a chain of CFM points

6.2.4 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.

6.2.5 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.

6.3 Compile-Time Cost-Benefit Analysis of Dynamic Predication

In the basic algorithms presented in Section 6.2 (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.[3]

[3] 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.

6.3.1 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:

$$\begin{aligned} dpred\_cost = dpred\_overhead * P(enter\_dpred\_corr\_pred) \\ +(dpred\_overhead - misp\_penalty) * P(enter\_dpred\_misp) \end{aligned} \tag{6.1}$$

$$P(enter\_dpred\_corr\_pred) = 1 - Acc\_Conf \tag{6.2}$$

$$P(enter\_dpred\_misp) = Acc\_Conf \tag{6.3}$$

$dpred\_overhead$: 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, and

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 (6.1), is less than zero (i.e., if the benefit of dynamic predication is positive in terms of execution cycles):

$$\text{Select a branch as a diverge branch if } dpred\_cost < 0 \tag{6.4}$$

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 [35]. Confidence estimator accuracy (defined as the percentage of low-confidence branches that are actually mispredicted, i.e., PVN [30]) is usually between 15%-50% and is dependent on confidence estimator parameters such as the threshold values used in the design [30]. 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.[4]

6.3.1.1 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 candi-

[4] 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%).

date 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 6.5-6.11 show how the compiler calculates $N(dpred\_insts)$ with these three different methods using the example presented in Figure 5.1. Note that the most frequently executed paths are shaded in Figure 5.1. 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.[5] In this chapter, we evaluate methods 2 and 3.

$$N(dpred\_insts) \quad = \quad N(BH) + N(CH) \tag{6.5}$$

$N(BH)$: 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

[5] 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.

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

$$\begin{aligned} N(BH) &= N(B) + N(E) && (6.6)\\ N(CH) &= N(C) && (6.7) \end{aligned}$$

(Method 2) Based on the longest possible path:

$$\begin{aligned} N(BH) &= MAX\{N(B) + N(D) + N(F), \\ &\quad N(B) + N(D) + N(E), N(B) + N(E)\} && (6.8)\\ N(CH) &= N(C) + N(G) && (6.9) \end{aligned}$$

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

$$\begin{aligned} N(BH) &= N(B) + P(BE) * N(E) + P(BD) * P(DE) * N(E) \\ &\quad + P(BD) * N(D) + P(BD) * P(DF) * N(F) && (6.10)\\ N(CH) &= N(C) + P(CG) * N(G) && (6.11) \end{aligned}$$

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.

$$N(useful\_dpred\_insts) = P(AB) * N(BH) + P(AC) * N(CH) \quad (6.12)$$

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

$$N(useless\_dpred\_insts) = N(dpred\_insts) - N(useful\_dpred\_insts) \quad (6.13)$$

$$dpred\_overhead = N(useless\_dpred\_insts)/fw \quad (6.14)$$

fw: Machine-specific instruction fetch width

$useless\_dpred\_insts$: Useless instructions fetched during dpred-mode

Combining Equation (6.14) with Equations (6.1) and (6.4) 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

$$\{(N(useless\_dpred\_insts)/fw) - misp\_penalty\} * P(enter\_dpred\_misp) + \{N(useless\_dpred\_insts)/fw\} * P(enter\_dpred\_corr\_pred) < 0 \quad (6.15)$$

6.3.2 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 (6.14). Therefore, we use the following equation to calculate the dynamic predication overhead of a frequently-hammock:

$$\begin{aligned} dpred\_overhead = \{1 - P(merge)\} * \{branch\_resol\_cycles/2\} + \\ P(merge) * \{N(useless\_dpred\_insts)/fw\} \end{aligned} \tag{6.16}$$

$P(merge)$: 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 (6.1) and (6.4) to determine whether or not selecting a frequently-hammock branch as a diverge branch would be beneficial for performance.

6.3.3 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 6.2.3.1, the overhead of dynamically predicating a diverge branch with multiple CFM points is computed assuming all CFM points (X_i) are independent:

$$\begin{aligned} &dpred\_overhead = \\ &\quad \{\sum_i N(useless\_dpred\_insts(X_i)) * P(merge\ at\ X_i)\}/fw + \\ &\quad \{1 - \sum_i P(merge\ at\ X_i)\} * \{branch\_resolution\_cycles/2\} \end{aligned} \tag{6.17}$$

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

If the diverge branch candidate satisfies Equations (6.1) and (6.4) after using the $dpred\_overhead$ developed in Equation (6.17), the branch is selected as a diverge branch with its reduced list of CFM points.

6.3.4 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 1 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.

6.4 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. 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.

6.4.1 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:

$$dpred\_overhead = N(select\_uops) * dpred\_iter/fw \tag{6.18}$$

$N(select\_uops)$: 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 (6.18)).

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:

$$\begin{aligned} dpred\_overhead \;&=\; N(loop\_body) * dpred\_extra\_iter/fw + \\ &\quad\; N(select\_uops) * dpred\_iter/fw \end{aligned} \tag{6.19}$$

$N(loop\_body)$: 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 (6.18).

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

$$\begin{aligned} dpred\_cost = \ & dpred\_overhead(corr\_pred) * P(enter\_dpred\_corr\_pred) \\ & + dpred\_overhead(early\_exit) * P(early\_exit) \\ & + dpred\_overhead(late\_exit) * P(late\_exit) \\ & + dpred\_overhead(no\_exit) * P(no\_exit) \\ & - misp\_penalty * P(late\_exit) \end{aligned} \tag{6.20}$$

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

6.4.2 Heuristics to Select Diverge Loop Branches

According to the cost-benefit model presented in Section 6.4.1, 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 (6.19).
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 this chapter, we use $STATIC\_LOOP\_SIZE = 30$, $DYNAMIC\_LOOP\_SIZE = 80$, and $LOOP\_ITER = 15$, which we empirically determined to provide the best performance.

6.5 Methodology

6.5.1 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 6.2, and evaluate the diverge branch candidates using the cost-benefit model developed in Sections 6.3 and 6.4. 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.

6.5.2 Simulation Methodology

Simulation Methodology is described in Section 5.4. The benchmarks are run to completion with a reduced input set [46] to reduce simulation time. Section 6.6.3 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.

6.6 Results

6.6.1 Diverge Branch Selection Algorithms

Figure 6.2 and 6.3 show the performance improvement of DMP with different diverge branch selection algorithms. Figure 6.2 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).[6] All algorithms use thresholds that are empirically

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

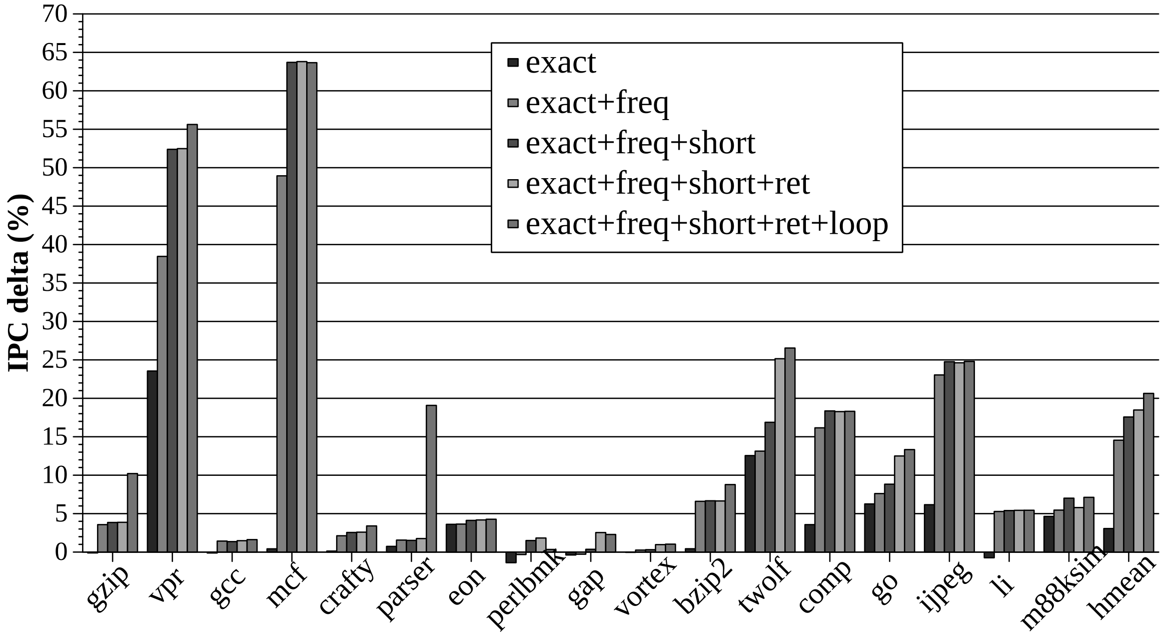


Figure 6.2: Performance improvement of DMP with Alg-exact and Alg-freq selection algorithms

determined to provide the best performance.

According to Figure 6.2 the performance benefit of DMP increases as we cumulatively employ our diverge branch selection techniques. Using just Alg-exact, DMP provides a performance improvement of 3.1%. However, when all our techniques are used, the performance improvement of DMP increases to 20.6%. Figure 6.4 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 6.2, selecting frequently-hammocks (Alg-freq) improves average performance by 11% on top of Alg-exact. Hence, the selection of frequently-hammocks

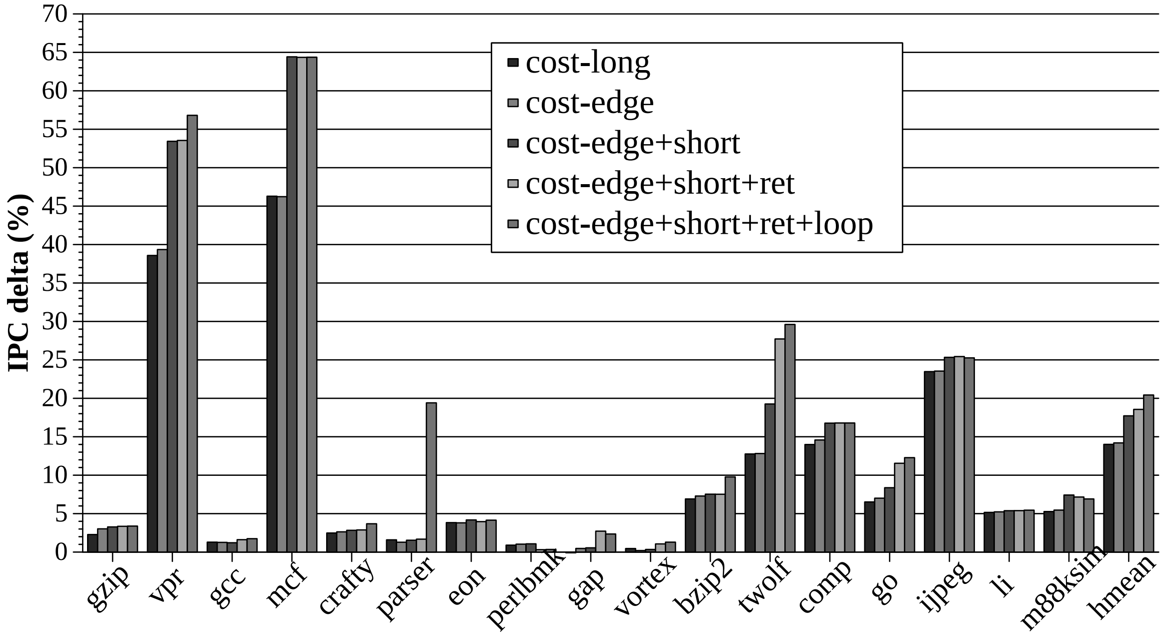


Figure 6.3: Performance improvement of DMP with cost-benefit analysis based selection algorithms

is the largest contributor to the performance of dynamic predication. Always predicating short hammocks improves performance by 3.0% on average and by more than 4% in vpr (14%), mcf (15%) 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 15% performance benefit. Including return CFM points improves performance by 0.9% on average and by more than 3% in twolf (8.3%) and go (3.6%). 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 6.4 provides an additional 2.2% average performance improvement, especially in gzip (6.3%) and parser (17%). Parser has a frequently-executed small loop in which an input word is compared to

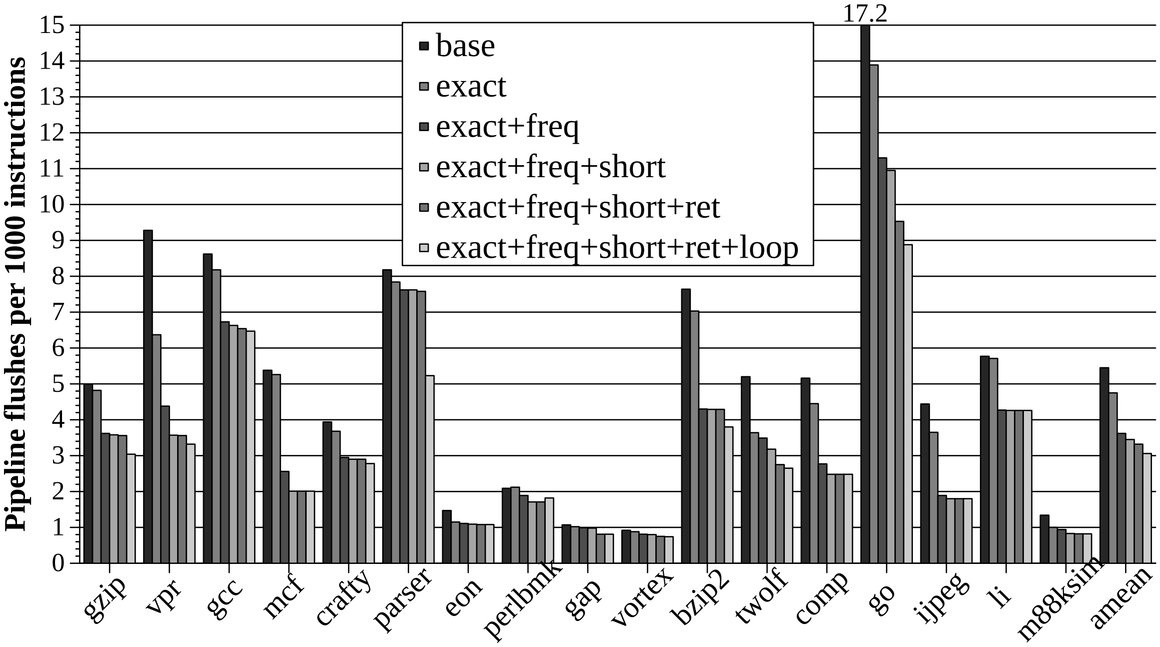


Figure 6.4: Pipeline flushes due to branch mispredictions in the baseline and DMP

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.

Figure 6.3 shows the performance improvement of DMP if we use the cost-benefit analysis developed in Section 6.3 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 6.3.1.1, and edge-profile-based average path (cost-edge), method 3 in Section 6.3.1.1. 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 6.2 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).[7] Using all these optimizations in conjunction with cost-edge results in 20.4% 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.

6.6.1.1 Effect of Optimizing Branch Selection Thresholds

Figure 6.5 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 3.7%. Therefore, optimizing the thresholds used in our heuristic-based selection algorithms is important to obtain the best performance. This observation also argues for

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

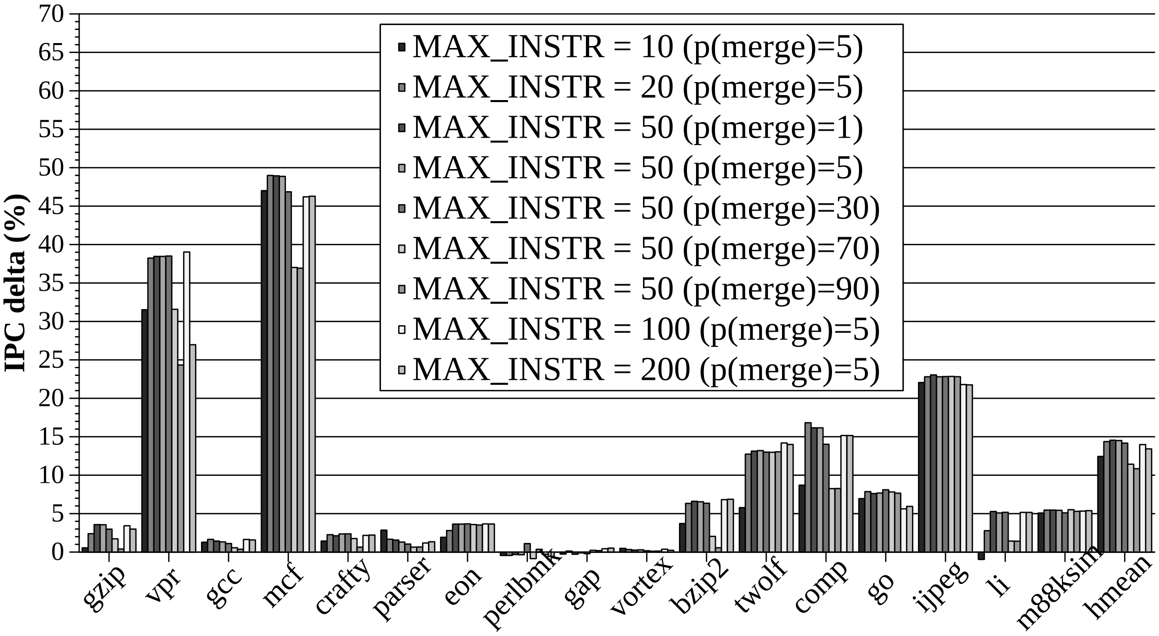


Figure 6.5: Performance improvement of DMP with different MIN_MERGE_PROB and MAX_INSTR heuristics

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 6.5 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.

6.6.2 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 6.6 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.[8] 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 6.6 shows that *Every-br*, *High-BP-5*, and *Immediate* are the best-performing simple algorithms for selecting diverge branches with average performance improvements of 6.5%, 4.3% 6.4% respectively. However, none of these other algorithms provide as large performance improvements as our technique, which improves average performance by 20.6%. 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 (*Every-br*) perform almost as well as our scheme. Gcc has very complex CFGs

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

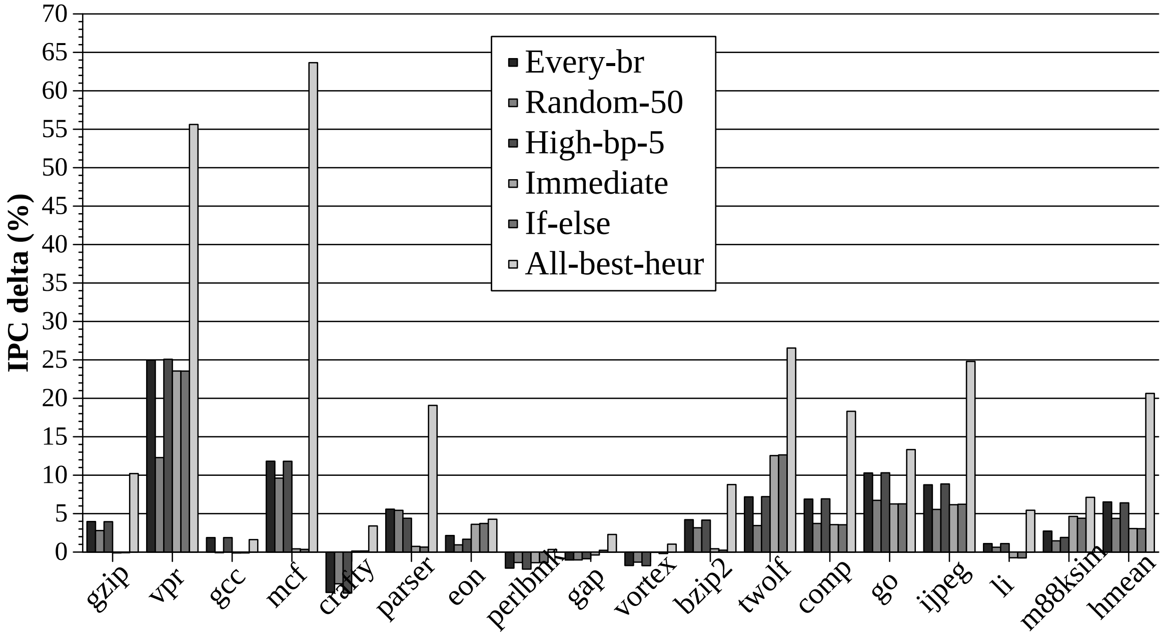


Figure 6.6: Performance improvement of DMP with alternative simple algorithms for selecting diverge branches

(that usually do not result in frequently-hammocks), so there are few diverge branch candidates. Gcc also has a very high branch misprediction rate (5%). *Every-br* allows the processor to enter dpred-mode for *all low-confidence branches*, which covers 50% 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 23% of all mispredicted branches.

6.6.3 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 6.7 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*).

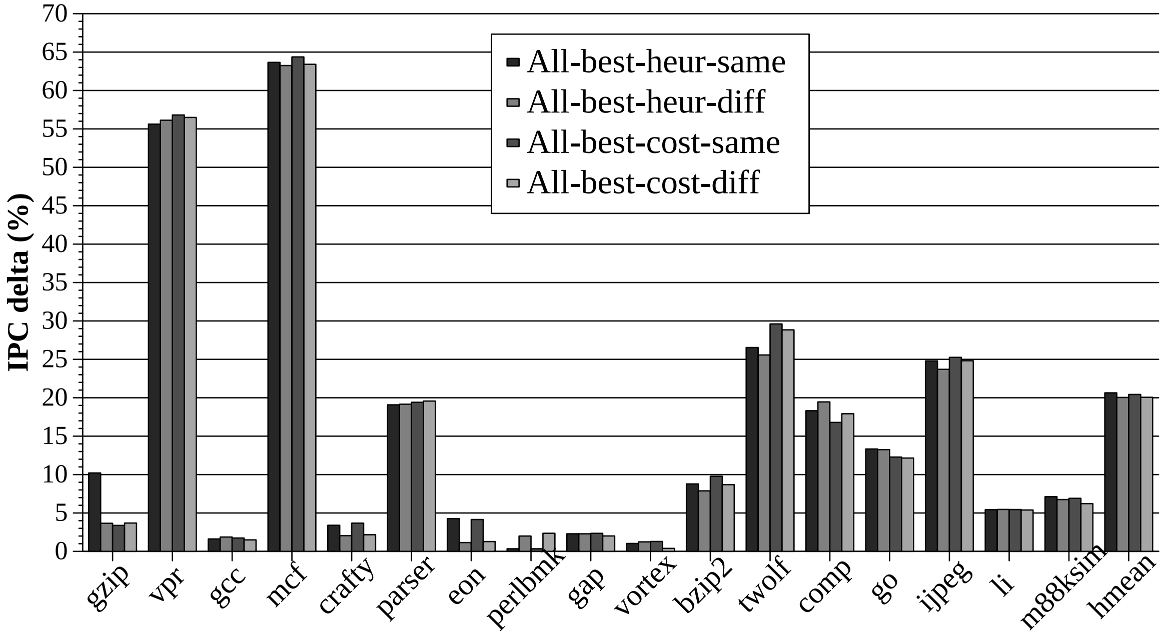


Figure 6.7: Performance improvement of DMP when a different input set is used for profiling

Figure 6.7 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 6.4%) 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.

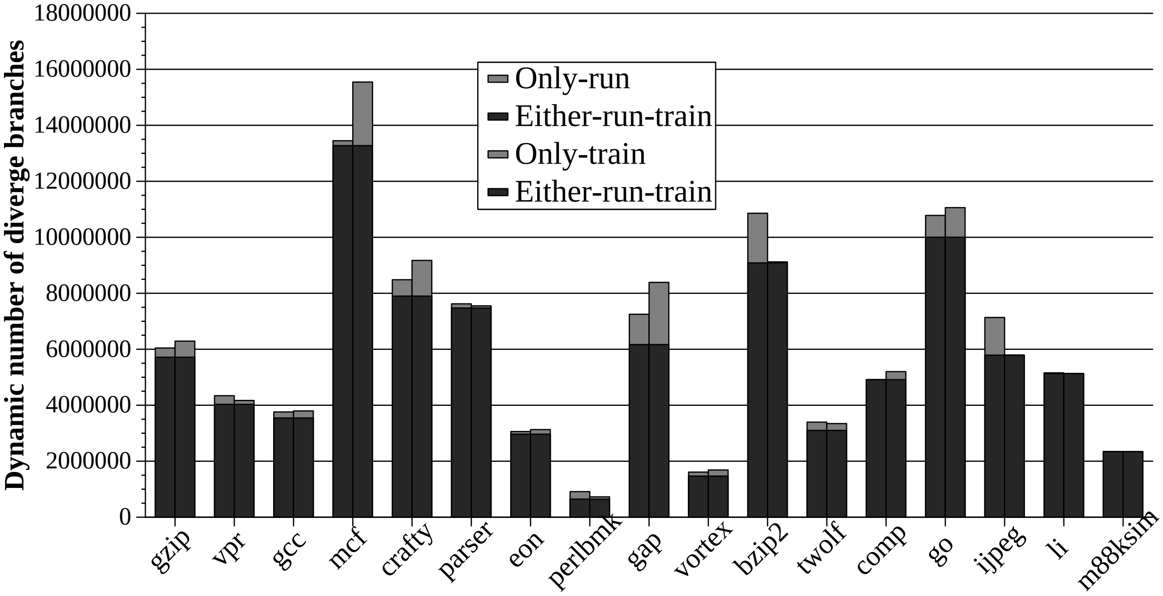


Figure 6.8: Dynamic diverge branches selected by different input sets (only run-time, only train, or either input). Left bar: profiling with run-time input, Right bar: profiling with train input

Figure 6.8 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 [46]) 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 6.8 show 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 [10], (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.

6.7 Summary

This chapter 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 20.6%. In contrast, the best-performing alternative branch selection algorithm results in a performance increase of only 4.5% over the baseline.

Chapter 7

Conclusions and Future Research Directions

7.1 Conclusions

Branch misprediction penalty is an important performance limiter and a major reason of wasted energy in high-performance processors. Predication has been used to avoid pipeline flushes due to branch mispredictions by converting control dependencies into data dependencies. However, predication has three major limitations/problems: adaptivity, complex CFG, and ISA, as Chapter 1 showed. This dissertation proposed and evaluated the adaptive predicated execution paradigm to solve these three limitations/problems.

The adaptive predicated execution paradigm provides a choice to the hardware: the choice of whether or not to use predicated execution for each dynamic instance of a branch instruction. This dissertation proposed two mechanisms to implement the adaptive predicated execution paradigm, wish branches and the diverge-merge processor architecture.

Chapter 4 proposed wish branches and evaluated the performance benefit of wish branches. Wish branches are a set of new control flow instructions, that combine both branch prediction and predicated execution. With wish branches, the compiler generates code that can be executed either as normal branch code or as predicated code. At run-time, the hardware chooses between normal branch code and predicated code based on the run-time branch behavior. Hence, wish branches provide adaptivity to predicated code to dynamically eliminate the overhead of predicated execution. Furthermore, wish loops provide a mechanism to exploit predicated execution to reduce the branch misprediction penalty for backward (loop) branches. The results in Chapter 4 show that wish branches improve the average execution time of nine SPEC INT 2000 benchmarks on an aggressive out-of-order superscalar processor by 14.2% compared to conditional branch prediction and by 13.3% compared to the best performing predicated code binary.

Although wish branches can provide the adaptivity to predicated code, wish branches still need the predicated ISA support. Furthermore, the compiler cannot convert most complex CFGs to wish branches because wish branch code is generated at static (compilation) time. Hence, to enable adaptive predicated execution in non-predicated ISA and to overcome the complex CFG problem of software predicated execution, Chapter 5 proposed the diverge-merge processor (DMP) and evaluated its performance benefit.

In DMP, instead the compiler produces a predicated version of code, the processor dynamically predicates instructions. The compiler provides control-flow information (a diverge branch and the corresponding control-flow merge point) to simplify the hardware used for dynamically predicating the code. If a diverge branch is hard-to-predict at run-time, the processor dynamically predicates the instructions between the diverge branch and the control-flow merge point. Hence, hard-to-predict branches can be executed as predicated code at run-time without requiring full support for predication in the ISA. The diverge-merge processor can dynamically predicate a branch if frequently executed paths of the branch look/behave like a simple hammock even though the control flow graph is not a really hammock. Hence, DMP can also overcome the complex CFG problem. The results showed that about 66% of dynamic mispredicted branches can be dynamically predicated in DMP.

Chapter 5 also compared DMP with five major previously-proposed branch processing paradigms, both qualitatively in terms of functionality and complexity and quantitatively in terms of performance benefits and energy/power consumption. DMP is able to predicate a much larger set of CFGs that cause mispredictions than dynamic hammock predication, software predication, wish branches, and dual-path execution because DMP enables the predication of frequently-hammocks. The results showed that DMP has much less overhead than dual-path/multipath execution paradigms because DMP does not execute control-independent instructions multiple times. Therefore, the average IPC improvement over all benchmarks is 3.5% for dynamic hammock predication, 4.8% for dual-path, 8.8% for multipath, and 19.3% for DMP. Conventional software predication reduces execution time by 3.8%, wish branches by 6.4%, and DMP by 13.0%. DMP provides the best energy efficiency and energy-delay product (EDP) among all paradigms, reducing energy consumption by 9% and improving EDP by 22.3% due to a 38% reduction in pipeline flushes. Even on a less aggressive processor with a short pipeline and a small instruction window, DMP improves performance by 7.8% while improving EDP by 9.7%.

Finally, the dissertation also presented the code generation algorithms for DMP architecture in Chapter 6. The algorithms select branches that are suitable and profitable for dynamic predication based on profiling information and corresponding control-flow merge (CFM) points. We also developed a new profile-driven analytical cost-benefit model to select branches that are profitable for dynamic predication.

Based on the results presented in this dissertation, we believe that the adaptive predicated execution has three major advantages:

1. The adaptive predicated execution overcomes the three major problems/limitations of predicated execution: adaptivity, complex CFG and ISA.
2. Wish branches in Chapter 4 provide the hardware with a choice to use branch prediction or predicated execution for each dynamic instance of a branch.
3. DMP in Chapter 5 eliminates branch misprediction flushes much more efficiently (i.e., with less instruction execution overhead) than alternative approaches, especially dual-path and multipath execution.

Hence, we conclude that the adaptive predicated execution paradigm provides a high performance and energy efficient mechanism to reduce the branch misprediction penalty.

7.2 Future Research Directions

7.2.1 Wish Branch Generation Algorithms

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

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

Similarly, if the compiler can take into account the execution delay due to the data dependencies on predicates when estimating the execution time of wish branch code on an out-of-order processor, it can perform a more accurate cost-benefit analysis to determine what to do with a branch. Such heuristics will also be useful in generating better predicated code for out-of-order execution processors.

7.2.2 Diverge-Merge Processor

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 the future work.

The cost of implementing the diverge-merge processor could be reduced in other processing paradigms such as Simultaneous Multithreading (SMT). SMT processors already support multiple fetch mechanisms and multiple active renaming mechanisms, which will reduce the cost of implementing DMP.

On the compiler side, future research can focus on the exploration of more accurate cost-benefit models. In particular, the proposed cost model for loop diverge branches in Chapter 6 requires the profiler to collect DMP-specific information. It is worth while to examine techniques that can make the cost model for selecting loop branches implementable. Besides static cost-benefit models, 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.

Appendix

Appendix A

Input Dependent Branches

One of the motivations of this dissertation is that branch misprediction rate changes depending on an input to a program, program phase [69, 72], and a control-path [13, 12] that leads to a branch. Sherwood and Calder [69] showed that the average program's branch misprediction rate has time varying behavior. Chappell [13, 12] quantitatively analyzed branch misprediction rate characteristics depending on program paths. However, not many researchers have shown how much individual branch's misprediction rate is dependent on input sets. Hence, this appendix discusses input dependent branches.

A.1 Input Dependent Branches

We classify a conditional branch as input-dependent if its prediction accuracy changes by a certain threshold value across two input sets. We set this threshold to be 5% in our analysis. For example, if the prediction accuracy of a branch instruction is 80% with one input set and 85.1% with another, this branch is considered to be an input-dependent branch since the delta, 5.1%, is greater than the threshold, 5%.

A.2 Frequency and Characteristics of Input-Dependent Branches

Figure A.1 shows the dynamic and static fraction of conditional branches that show input-dependent behavior. Train and reference input sets for the SPEC INT 2000 benchmarks were used to identify the input-dependent branches. Our baseline branch predictor is a 4KB gshare branch predictor. The dynamic fraction is obtained by dividing the number of dynamic instances of all input-dependent branches by the number of dynamic instances of all branch instructions, using the reference input set. The benchmarks are sorted by the dynamic fraction of input-dependent branches, in descending order from left to right.[1] The data shows that there are many branches that show more than 5% absolute change in prediction accuracy between the train and reference input sets. More than 10% of the static branches in bzip2, gzip, twolf, gap, crafty, and gcc are input-dependent branches. Note that this data is obtained using only two input sets to define the set of input-dependent branches.

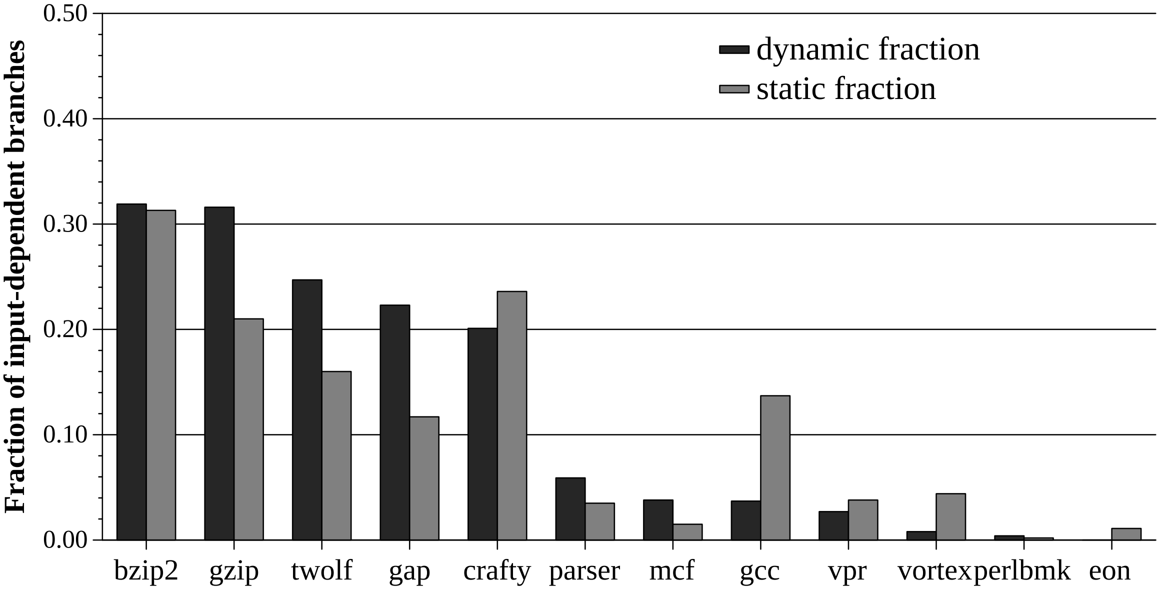


Figure A.1: The fraction of input-dependent branches (using train and reference input sets)

Figure A.2 shows whether or not all input-dependent branches are hard-to-predict.

[1] Note that input-dependence is a property of a *static* branch. Input-dependence cannot be defined for a dynamic instance of a branch, since the dynamic instance of a branch is executed only once. We show the dynamic fraction of input-dependent branches in Figure A.1 to provide insight into the execution frequency of input-dependent branches. All other results in this paper are based on *static branches*.

This figure displays the distribution of all input-dependent branches based on their prediction accuracy. Input-dependent branches are classified into six categories based on their prediction accuracy using the reference input set. The data shows that a sizable fraction of input-dependent branches are actually relatively easy-to-predict (i.e., have a prediction accuracy of greater than 95%) in many of the benchmarks. Even the fraction of input-dependent branches with a prediction accuracy greater than 99% -which is a very strict accuracy threshold- is significant for gap (19%), vortex (8%), gcc (7%), crafty (6%), twolf (4%), and parser (4%). Hence, not all input-dependent branches are hard-to-predict. There are many input-dependent branches that are relatively easy-to-predict.

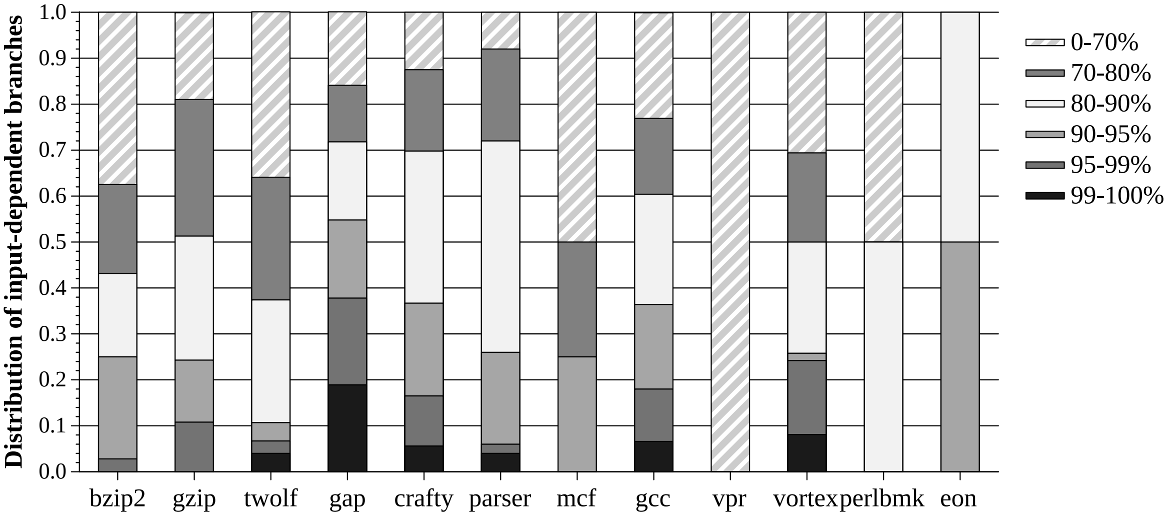


Figure A.2: The distribution of input-dependent branches based on their branch prediction accuracy

Figure A.3 shows whether or not all hard-to-predict branches are input-dependent. We classify all branches into six categories based on their prediction accuracy. The figure presents the fraction of input-dependent branches in each category. For example, in bzip2, 75% of branches with a prediction accuracy lower than 70% are input-dependent and only 10% of branches with a prediction accuracy between 95-99% are input-dependent.

In general, the fraction of input-dependent branches increases as the prediction accuracy decreases. Thus, branches with a low prediction accuracy are more likely to be input-dependent. However, many branches with a low prediction accuracy are actually not input-dependent. For example, in gzip only half of the branches with a prediction accuracy lower than 70% are input-dependent.

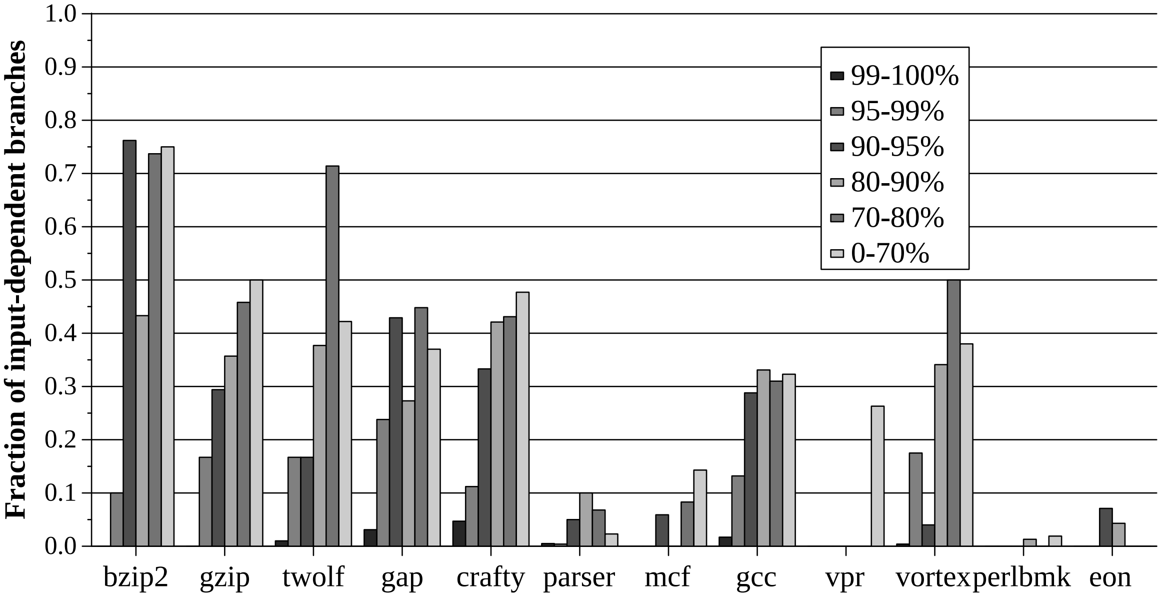


Figure A.3: The fraction of input-dependent branches in different prediction accuracy categories

We also measure the overall branch misprediction rate to examine the correlation between the overall branch misprediction rate difference across input sets and the fraction of input-dependent branches. Table A.1 shows the average branch misprediction rate for each input set. Some benchmarks that have a small difference in the overall branch misprediction rate between the two input sets, such as eon and perlbmk, also have a small fraction of input-dependent branches (as can be seen in Figure A.1). For these benchmarks, profiling with multiple input sets and computing the average branch prediction accuracy would correctly indicate that there are not many input-dependent branches. In contrast, even though twolf and crafty have a small difference in overall branch prediction accuracy across the two input sets, they have a high number of input-dependent branches. So, just comparing the overall branch misprediction rate across input sets does not provide enough information to judge whether or not a benchmark has many input-dependent branches.

Table A.1: Average branch misprediction rates of the evaluated programs (%)

Input Data Set	bzip2	gzip	twolf	gap	crafty	parser	mcf	gcc	vpr	vortex	perlbmk	eon
train	1.9	7.5	16.4	5.7	12.4	9.1	7.8	7.3	11.2	0.8	5.1	12.2
reference	8.3	6.5	15.7	3.9	11.8	8.9	6.6	2.4	11.1	0.4	5.1	12.1

A.3 Examples of Input-Dependent Branches

What kind of branches are sometimes easy to predict and sometimes hard to predict? We provide two examples to show the code structures causing input-dependent branch behavior.

One example of an input-dependent branch is a branch that checks data types. A branch in the gap benchmark, which is shown on line 5 in Figure A.4, checks whether or not the data type of a variable (hd) is an integer. The program executes different functions depending on the data type of the variable. The misprediction rate of this branch is 10% with the train input set, but it is 42% with the reference input set. With the train input set, the variable is an integer for 90% of the time, so the taken rate of the branch is 90%. Hence, even a simple predictor achieves 90% accuracy for that branch. In contrast, with the reference input set, approximately half of the time the variable is of non-integer type and therefore the branch misprediction rate increases to 42%. Gap is a math program that can compute using different types of data. It uses a non-integer data type to store values greater than 2^{30}. The reference input set contains a large fraction of values that are greater than 2^{30}, which are stored in variables of a non-integer data type. In contrast, most input data values in the train input set are smaller than 2^{30} and they are stored as integers. This results in very different behavior across input sets for the branch that checks the type of the input data.

```
1 :TypHandle Sum ( TypHandle hd ) {
2 :   // initialize hdL and hdR using hd
3 :   // ...
4 :   // input-dependent br. checks the type of hd (line 5)
5 :   if ( (long)hdL & (long)hdR & T_INT ) {
6 :        // use integer sum function for integer type
7 :        result = (long)hdL + (long)hdR - T_INT;
8 :        ov = (int)result;
9 :        if ( ((ov << 1) >> 1) == ov )
10:           return (TypHandle) ov; // return integer sum
11:   }
12:
13:  // call a special SUM function for non-integer type
14:  return SUM( hdL, hdR );
15:}
```

Figure A.4: An input-dependent branch from `gap`

The prediction behavior of a loop branch is strongly dependent on what determines the number of loop iterations. If the loop iteration count is determined by input data, the prediction behavior of the loop branch is dependent on the input set. If the iteration count is a large number, then the branch is easy to predict, whereas if the iteration count is small, the branch can be hard to predict. For example, some loops in the gzip benchmark execute for different number of iterations depending on the compression level, which is specified as a parameter to the program. Figure A.5 shows an example. The branch on line 25 is a loop exit branch. The exit condition is defined on line 18 using *pack_level* and *max_chain*. *pack_level* is the compression level and *max_chain* is the value that determines the number of loop iterations. *max_chain* has a higher value at higher compression levels, as shown on lines 9-13. At compression level 1, the loop iterates 4 times and the prediction accuracy of the branch is 75% (3/4) without a specialized loop predictor. But, at compression level 9, the loop iterates 4096 times, so the prediction accuracy of the branch is very close to 100% (4095/4096). Therefore, the branch is input-dependent on the input parameter that specifies the compression level.

```
1: typedef struct config {
2:    int good_length;
3:    int max_lazy;
4:    int nice_length;
5:    int max_chain;
6: } config;
7:
8: local config config_table[10] = {
9:   /* 1 */ {4,    4,   8,    4}, // min compression level
10:  // ...
11:  /* 4 */ {4,    4,  16,   16},
12:  // ...
13:  /* 9 */ {32, 258, 258, 4096} // max compression level
14: };
15:
16: /*** Initialization code begin ***/
17:   // max chain length is read from the config table
18:   max_chain_length = config_table[pack_level].max_chain;
19:   unsigned chain_length = max_chain_length;
19: /*** Initialization code end ***/
20:
21: do {
22:    // ...
23:    // input-dependent loop exit branch (line 25)
24: } while ((cur_match = prev[cur_match & WMASK]) > limit
25:             && --chain_length != 0);
```

Figure A.5: An input-dependent loop exit branch from `gzip`

Bibliography

[1] P. S. Ahuja, K. Skadron, M. Martonosi, and D. W. Clark. Multipath execution: opportunities and limits. In *Proceedings of the 12th International Conference on Supercomputing*, pages 101–108, 1998.

[2] H. Akkary, S. T. Srinivasan, R. Koltur, Y. Patil1, and W. Refaai. Perceptron-based branch confidence estimation. In *10th International Symposium on High Performance Computer Architecture (HPCA)*, pages 265–275, 2004.

[3] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *10th ACM Symposium on Principles of Programming Languages*, pages 177–189, 1983.

[4] D. Anderson, F. Sparacio, and R. Tomasulo. The IBM system/360 model 91: Machine philosophy and instruction-handling. *IBM Journal of Research and Development*, 11(1):8–24, Jan. 1967.

[5] 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 *Proceedings of the Third IEEE International Symposium on High Performance Computer Architecture*, pages 84–93, 1997.

[6] D. I. August, W. W. Hwu, and S. A. Mahlke. A framework for balancing control flow and predication. In *Proceedings of the 30th ACM/IEEE International Symposium on Microarchitecture*, pages 92–103, 1997.

[7] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 83 – 94, 2000.

[8] M. D. Brown, J. Stark, and Y. N. Patt. Select-free scheduling logic. In *Proceedings of the 34th ACM/IEEE International Symposium on Microarchitecture*, pages 204–213, 2001.

[9] P.-Y. Chang, E. Hao, and Y. N. Patt. Alternative implementations of hybrid branch predictors. In *Proceedings of the 28th ACM/IEEE International Symposium on Microarchitecture*, pages 252–263, 1995.

[10] 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 *Proceedings of the 1995 ACM/IEEE Conference on Parallel Architectures and Compilation Techniques*, pages 99—108, 1995.

[11] R. Chappell, P. Racunas, F. Tseng, S. Kim, M. Brown, O. Mutlu, H. Kim, M. Qureshi, J. A. Joao, and C. J. Lee. The scarab microarchitectural simulator. Unpublished documentation.

[12] R. S. Chappell. *Simultaneous Subordinate Microthreading (SSMT)*. PhD thesis, University of Michigan, 2004.

[13] R. S. Chappell, F. Tseng, A. Yoaz, and Y. N. Patt. Difficult-path branch prediction using subordinate microthreads. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 307—317, 2002.

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

[15] C.-Y. Cher and T. N. Vijaykumar. Skipper: a microarchitecture for exploiting control-flow independence. In *Proceedings of the 34th ACM/IEEE International Symposium on Microarchitecture*, pages 4–15, 2001.

[16] 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 *Proceedings of the 34th ACM/IEEE International Symposium on Microarchitecture*, pages 182–191, 2001.

[17] Y. Chou, B. Fahs, and S. Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. In *Proceedings of the 31th International Symposium on Computer Architecture*, pages 76—87, 2004.

[18] Y. Chou, J. Fung, and J. P. Shen. Reducing branch misprediction penalties via dynamic control independence detection. In *Proceedings of the 13th International Conference on Supercomputing*, pages 109–118, 1999.

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

[20] J. D. Collins, D. M. Tullsen, and H. Wang. Control flow optimization via dynamic reconvergence prediction. In *Proceedings of the 37th ACM/IEEE International Symposium on Microarchitecture*, pages 129–140, 2004.

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

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

[23] 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):48–57, May 2005.

[24] 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.

[25] R. R. dos Santos. *DCE: The Dynamic Conditional Execution in a Multipath Control Independent Architecture*. PhD thesis, PPGC/UFRGS, 2003.

[26] R. R. dos Santos, T. G. S. dos Santos, M. L. Pilla, P. O. A. Navaux, S. Bampi, and M. Nemirovsky. Complex branch profiling for dynamic conditional execution. In *Proceedings of the 15th Symposium on Computer Architecture and High Performance Computing*, pages 28—35, 2003.

[27] 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.

[28] A. Gandhi, H. Akkary, and S. T. Srinivasan. Reducing branch misprediction penalty via selective recovery. In *Proceedings of the Tenth IEEE International Symposium on High Performance Computer Architecture*, pages 254–264, 2004.

[29] GCC-4.0. GNU compiler collection. http://gcc.gnu.org/.

[30] D. Grunwald, A. Klauser, S. Manne, and A. Pleszkun. Confidence estimation for speculation control. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 122–131, 1998.

[31] K. Hazelwood and T. Conte. A lightweight algorithm for dynamic if-conversion during dynamic optimization. In *Proceedings of the 2000 ACM/IEEE International Conference on Parallel Architectures and Compilation Techniques*, 2000.

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

[33] P. Hsu and E. Davidson. Highly concurrent scalar processing. In *Proceedings of the 13th International Symposium on Computer Architecture,* pages 386–395, 1986.

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

[35] E. Jacobsen, E. Rotenberg, and J. E. Smith. Assigning confidence to conditional branch predictions. In *Proceedings of the 29th ACM/IEEE International Symposium on Microarchitecture,* pages 142–152, 1996.

[36] D. A. Jiménez and C. Lin. Dynamic branch prediction with perceptrons. In *Proceedings of the Seventh IEEE International Symposium on High Performance Computer Architecture,* pages 197–206, 2001.

[37] D. A. Jiménez and C. Lin. Composite confidence estimators for enhanced speculation control. Technical report, Department of Computer Sciences, The University of Texas at Austin, Jan. 2002.

[38] R. Ju, S. Chan, C. Wu, R. Lian, and T. Tuo. Open research compiler for Itanium processor family. In *MICRO-34 Tutorial*, 2001.

[39] 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 *Proceedings of the 39th ACM/IEEE International Symposium on Microarchitecture*, pages 53–64, 2006.

[40] H. Kim, J. A. Joao, O. Mutlu, and Y. N. Patt. Profile-assisted compiler support for dyanmic predication in diverge-merge processors. In *Proceedings of the Fifth International Symposium on Code Generation and Optimization*, pages 367–378, 2007.

[41] H. Kim, O. Mutlu, J. Stark, and Y. N. Patt. Wish branches: Combining conditional branching and predication for adaptive predicated execution. In *Proceedings of the 38th ACM/IEEE International Symposium on Microarchitecture*, pages 43–54, 2005.

[42] H. Kim, M. A. Suleman, O. Mutlu, and Y. N. Patt. 2D-profiling: Detecting input-dependent branches with a single input data set. In *Proceedings of the Fourth International Symposium on Code Generation and Optimization*, pages 159–172, 2006.

[43] A. Klauser, T. Austin, D. Grunwald, and B. Calder. Dynamic hammock predication for non-predicated instruction set architectures. In *Proceedings of the 1998 ACM/IEEE Conference on Parallel Architectures and Compilation Techniques*, pages 278–285, 1998.

[44] A. Klauser and D. Grunwald. Instruction fetch mechanisms for multipath execution processors. In *Proceedings of the 32nd ACM/IEEE International Symposium on Microarchitecture*, pages 38–47, 1999.

[45] A. Klauser, A. Paithankar, and D. Grunwald. Selective eager execution on the polypath architecture. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 250–259, 1998.

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

[47] A. K.Uht. *Multipath Execution*. CRC PRESS, 2005.

[48] Y. Liu, Z. Zhang, R. Qiao, and R. Ju. A region-based compilation infrastructure. In *Proc. of the 7th Workshop on Interaction between Compilers and Computer Architecture*, pages 75–84, 2003.

[49] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation*, pages 190–200, 2005.

[50] 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 *Proceedings of the 27th ACM/IEEE International Symposium on Microarchitecture*, pages 217–227, 1994.

[51] 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 *Proceedings of the 25th ACM/IEEE International Symposium on Microarchitecture*, pages 45–54, 1992.

[52] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: Speculation control for energy reduction. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 132–141, 1998.

[53] S. Mantripragada and A. Nicolau. Using profiling to reduce branch misprediction costs on a dynamically scheduled processor. In *Proceedings of the 14th International Conference on Supercomputing*, pages 206–214, 2000.

[54] S. McFarling. Combining branch predictors. Technical Report TN-36, Digital Western Research Laboratory, June 1993.

[55] T. Y. Morad, U. C. Weiser, A. Kolodny, M. Valero, and E. Ayguadé. Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors. *Computer Architecture Letters*, 5(1), 2006.

[56] 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 *Proceedings of the Ninth IEEE International Symposium on High Performance Computer Architecture*, pages 129–140, 2003.

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

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

[59] Y. N. Patt, W. Hwu, and M. Shebanow. HPS, a new microarchitecture: Rationale and introduction. In *Proceedings of the 18th ACM/IEEE International Symposium on Microarchitecture*, pages 103–107, 1985.

[60] Y. N. Patt, S. W. Melvin, W. Hwu, and M. C. Shebanow. Critical issues regarding HPS, a high performance microarchitecture. In *Proceedings of the 18th ACM/IEEE International Symposium on Microarchitecture*, pages 109–116, 1985.

[61] D. N. Pnevmatikatos and G. S. Sohi. Guarded execution and dynamic branch prediction in dynamic ILP processors. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 120–129, 1994.

[62] E. Quiones, J.-M. Parcerisa, and A. Gonzalez. Selective predicate prediction for out-of-order processors. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 46–54, 2006.

[63] 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.

[64] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. E. Smith. Trace processors. In *Proceedings of the 30th ACM/IEEE International Symposium on Microarchitecture*, pages 138–148, 1997.

[65] E. Rotenberg, Q. Jacobson, and J. E. Smith. A study of control independence in superscalar processors. In *Proceedings of the Fifth IEEE International Symposium on High Performance Computer Architecture*, pages 115–124, 1999.

[66] E. Rotenberg and J. Smith. Control independence in trace processors. In *Proceedings of the 32nd ACM/IEEE International Symposium on Microarchitecture*, pages 4–15, 1999.

[67] R. M. Russell. The CRAY-1 computer system. *Communications of the ACM*, 21(1):63–72, Jan. 1978.

[68] A. Seznec. Analysis of the O-GEometric History Length branch predictor. In *Proceedings of the 32th International Symposium on Computer Architecture*, pages 394–405, 2005.

[69] T. Sherwood and B. Calder. Time varying behavior of programs. Technical Report UCSD-CS99-630, UC San Diego, 1999.

[70] 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 *Proceedings of the 31th International Symposium on Computer Architecture*, pages 26–37, 2004.

[71] B. Simon, B. Calder, and J. Ferrante. Incorporating predicate information into branch predictors. In *Proceedings of the Ninth IEEE International Symposium on High Performance Computer Architecture*, pages 53–64, 2003.

[72] M. D. Smith. Overcoming the challenges to feedback-directed optimization. In *ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, pages 1–11, 2000.

[73] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 25–34, 2002.

[74] E. Sprangle and Y. Patt. Facilitating superscalar processing via a combined static/dynamic register renaming scheme. In *Proceedings of the 27th ACM/IEEE International Symposium on Microarchitecture*, pages 143–147, 1994.

[75] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual flow pipelines. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XI)*, pages 107–119, 2004.

[76] J. Stark, M. D. Brown, and Y. N. Patt. On pipelining dynamic instruction scheduling logic. In *Proceedings of the 33rd ACM/IEEE International Symposium on Microarchitecture*, pages 57–66, 2000.

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

[78] G. S. Tyson. The effects of predication on branch prediction. In *Proceedings of the 27th ACM/IEEE International Symposium on Microarchitecture*, pages 196–206, 1994.

[79] 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 *Proceedings of the*

Seventh IEEE International Symposium on High Performance Computer Architecture, pages 15–25, 2001.

[80] N. J. Warter, D. Lavery, and W. W. Hwu. The benefit of predicated execution for software pipelining. In *Proceedings of the Twenty-Sixth Hawaii International Conference on System Sciences*, pages 496–506, 1993.

[81] N. J. Warter, S. A. Mahlke, W. W. Hwu, and B. R. Rau. Reverse if-conversion. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, pages 290–299, 1993.

[82] T.-Y. Yeh and Y. N. Patt. Two-level adaptive branch prediction. In *Proceedings of the 24th ACM/IEEE International Symposium on Microarchitecture*, pages 51–61, 1991.

[83] T.-Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 124–134, 1992.

Vita

Hyesoon Kim was born in Taejon city, south Korea on August 29, 1974, the daughter of Mr. Sunsam Kim and Ms. Sookja Lee. She graduated from Taejon Science High School. She received the Bachelor of Science degree in Mechanical Engineering from Korea Advanced Institute of Science and Technology (KAIST) in 1996. She received the Master of Science degree in Mechanical Engineering from Seoul National University in 1998. The following year she started working at Hyundai-Motor company as a researcher until she entered the Ph.D. program at the University of Texas at Austin in the fall of 2000. She received the Master of Science and Engineering degree in Electrical Engineering in 2003, after which she continued her Ph.D studies.

While in graduate school, she served as a teaching assistant for two semesters at the University of Texas at Austin. She had summer internships at Intel for four times. She has published papers in major computer architecture and compiler conferences and journals (MICRO, ISCA, HPCA, CGO, IEEE-TC, IJPP). Three of the papers she co-authored have been selected in a collection of the most industry-relevant computer architecture research papers by the *IEEE Micro* technical magazine in the years 2005 and 2006.

Permanent address: Sam-mu-ri APT 212-1901, Dunsang-dong, Seou-gu
Taejon, Korea

This dissertation was typeset with LaTeX[†] by the author.

[†] LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.