ABSTRACT

AL SHEIKH, RAMI MOHAMMAD. Control-Flow Decoupling: An Approach for Timely, Non-speculative Branching. (Under the direction of Dr. Eric Rotenberg.)

Mobile and PC/server class processor companies continue to roll out flagship core microarchitectures that are faster than their predecessors. Meanwhile placing more cores on a chip coupled with constant supply voltage puts per-core energy consumption at a premium. Hence, the challenge is to find future microarchitecture optimizations that not only increase performance but also conserve energy. Eliminating branch mispredictions – which waste both time and energy – is valuable in this respect.

In this work, we explore the control-flow landscape by characterizing mispredictions in four benchmark suites. We find that a third of mispredictions-per-1K-instructions (MPKI) come from what we call *separable* branches: branches with large control-dependent regions (not suitable for if-conversion), whose backward slices do not depend on their control-dependent instructions or have only a short dependence. We propose control-flow decoupling (CFD) to eradicate mispredictions of separable branches. The idea is to separate the loop containing the branch into two loops: the first contains only the branch’s predicate computation and the second contains the branch and its control-dependent instructions. The first loop communicates branch outcomes to the second loop through an architectural queue. Microarchitecturally, the queue resides in the fetch unit to drive timely, non-speculative fetching or skipping of successive dynamic instances of the control-dependent region.

Either the programmer or compiler can transform a loop for CFD, and we evaluate both. On a microarchitecture configured similar to Intel’s Sandy Bridge core, CFD increases performance by up to 55%, and reduces energy consumption by up to 49%. Moreover, for some applications, CFD is a necessary catalyst for future complexity-effective large-window architectures to tolerate memory latency.

© Copyright 2013 by Rami Mohammad Al Sheikh

All Rights Reserved

Control-Flow Decoupling: An Approach for Timely, Non-speculative Branching

by
Rami Mohammad Al Sheikh

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Engineering

Raleigh, North Carolina

2013

APPROVED BY:

Dr. Greg Byrd

Dr. Huiyang Zhou

Dr. Xiaosong Ma

Dr. Eric Rotenberg
Chair of Advisory Committee

DEDICATION

This dissertation is lovingly dedicated to my beloved parents, who have been without doubt my greatest motivation and inspiration. Their support, encouragement, and constant love have sustained me throughout my life.

BIOGRAPHY

Rami Al Sheikh was born in Jordan, on March 13, 1984. Rami received the B.S. degree in electrical and computer engineering from the Hashemite University (HU) in 2006 and the M.S. degree in computer engineering from Jordan University of Science and Technology (JUST) in 2008. In 2009, he joined the Department of Electrical and Computer Engineering at North Carolina State University as a PhD candidate, where he started his research in high-performance microarchitectures under the direction of Dr. Eric Rotenberg. During his PhD studies, Rami has done two internships at Intel, where he worked with talented people from the industry on the development of future commercial processors. His research focuses on computer architecture, high-performance microarchitecture, branch prediction, hardware/software co-design, performance modeling, cache replacement, and dynamic binary translation and optimization. Upon the completion of his PhD program, he will be starting his first full-time job as a Senior Hardware Engineer at Qualcomm's CPU Research and Development Group, in Raleigh, North Carolina. Rami is a member of Phi Kappa Phi and IEEE.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my beloved parents, brother and two sisters. Their boundless love and constant support throughout my entire life, helped shape my character and gave me the strength and patience to tackle the difficult aspects of life.

The influence of my adviser Dr. Eric Rotenberg is immeasurable. He taught me so many things about computer architecture, research, and creative thinking. His continuous support, enthusiasm and guidance kept me moving forward, especially in the toughest times.

I would like to acknowledge my advisory committee members, Dr. Gregory Byrd, Dr. Huiyang Zhou and Dr. Xiaosong Ma, for their invaluable, constructive feedback after my preliminary examination, which helped improve the quality of my dissertation.

My utmost gratitude to Dr. James Tuck, without whom the compiler support would have never been possible. His contributions to the compiler implementation are countless, and deeply appreciated. My gratitude is beyond words.

Many thanks to Muawya Al-Otoom, Ahmad Samih, and Amro Awad, for the great times we had together, and for the intellectually stimulating discussions. I can never ask for better company.

I would like to extend special thanks to Mark Dechene for helping me during the early stages of my work. Also, I want to thank his wife, Monika, for the delicious food she used to make us.

Many thanks to Julian Taylor with whom I have had the pleasure of collaborating during my first year in the PhD program.

I would like to thank all current and former members of our research group and lab mates for their friendship and valuable discussions. In particular, Sandeep Navada, Elliott Forbes, Niket Choudhary, Salil Wadhavkar, Brandon Dwiell, Sahil Sabharwal, Rangeen Chowdhury, Jayneel Gandhi, Hashem Hashemi, Sounder Rajan Vijaya Kumar, Ashlesha Shastri, Hiran Mayukh, Tanmay Shah, Reetika Ganjoo, Saurabh Gupta, Anil Krishna, Siddhartha Chhabra, Avik Juneja, Devesh Tiwari, Ganesh Balakrishnan, and Yi Yang.

I am very thankful to Linda Fontes, Elaine Hardin, Katy Wilson, Joanne Pope-Clark, Jennifer Raab and Kendall Del Rio for providing exceptional administrative help.

I would like to acknowledge some of the industry folks for their help during my internships. From the binary translation team, Suresh Srinivas, Paul Caprioli, Jayaram Bobba, Jeffrey Cook, and Omar Shaikh. From the architecture team, Konrad Lai, Jared Stark, Hongliang Gao, and Beeman Strong.

This research was supported in part by Intel and NSF grant CCF-0916481. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Finally, I praise God who helped me complete this dissertation.

TABLE OF CONTENTS

LIST OF TABLES	viii
LIST OF FIGURES	ix
Chapter 1 Introduction	1
Chapter 2 Methodology and Control-Flow Classification	9
2.1 Methodology	9
2.2 Control-Flow Classification	11
Chapter 3 Control-Flow Decoupling	16
3.1 ISA Support and Benchmark Example	19
3.2 Software Side	22
3.3 Hardware Side	22
3.3.1 Early Push	23
3.3.2 Late Push	24
3.3.3 BQ Length	25
3.3.4 BQ Recovery	26
3.3.5 Branch Target Buffer	27
3.4 Optimizations	27
3.4.1 CFD+ (Value Queue)	28
3.4.2 DFD (Prefetching)	29
3.5 Separable Loop-branches	30
3.5.1 ISA Support and Benchmark Example	32
3.5.2 Software Side	32
3.5.3 Hardware Side	33
3.5.4 Support for Exceeding Maximum Trip-Count	34
3.6 Handling Complex Scenarios	36
3.6.1 Nested Hard-to-predict Separable Branches	36
3.6.1.1 Multi-level Decoupling	36
3.6.1.2 Software Speculation	36
3.6.2 Breaks	39
3.6.3 Support for Aggressively Reordering Loads and Stores	41
3.6.4 Decoupling Separable Branches at Different Loop-nest Levels	41
3.6.5 Bulk-push	43
Chapter 4 CFD Compiler Implementation	44
Chapter 5 Vectorization for Inseparable Branches	49
Chapter 6 Evaluation Environment	52

Chapter 7 Results and Analysis	55
7.1 Control-Flow Decoupling	59
7.1.1 CFD and CFD+	59
7.1.2 ASTAR Case Study	67
7.1.3 DFD	70
7.1.4 Automated CFD	73
7.1.5 Trip-count Queue	78
7.2 Vectorization	80
7.3 If-Conversion	82
7.4 Final Misprediction Breakdown	82
Chapter 8 Related Work	84
8.1 Static Branch Prediction	84
8.2 Dynamic Branch Prediction	85
8.2.1 Anti-aliasing Predictors	85
8.2.2 Long-history Predictors	86
8.2.3 Load-based Branch Predictors	87
8.2.4 Value-based Branch Predictors	88
8.2.5 Loop-branch Predictors	88
8.3 Branch Predication	89
8.4 Branch Pre-execution	90
8.4.1 Pre-execution Using Helper Threads	90
8.4.2 Pre-execution Leveraging Predicate Register File	91
8.5 Static Scheduling	92
8.6 Decoupled Architectures	92
8.7 Decoupled Software Pipelining	93
Chapter 9 Conclusion and Future Work	94
9.1 Conclusion	94
9.2 Future Work	95
References	97
Appendices	103
Appendix A Workloads with Predictable Control-Flow	104
Appendix B Properties of the Control-Flow Constrained Applications	106
Appendix C Branch Queue Size Selection	108
Appendix D Design Space Exploration for Baseline	111
Appendix E Applying CFD to Predictable Branches	115

LIST OF TABLES

Table 2.1	Targeted applications.	11
Table 6.1	Minimum fetch-to-execute latency in cycles.	53
Table 6.2	Baseline core configuration.	54
Table 6.3	Detailed storage overheads for BQ, VQ renamer, and TQ.	54
Table 7.1	CFD(BQ) and DFD application skip distances and overheads.	56
Table 7.2	CFD(TQ) application skip distances and overheads.	56
Table 7.3	VEC application skip distances and overheads.	57
Table 7.4	If-conversion application skip distances and overheads.	57
Table 7.5	Details of modified code for the CFD(BQ) applications.	57
Table 7.6	Details of modified code for the CFD(TQ) applications.	58
Table 7.7	Details of modified code for the VEC applications.	58
Table 7.8	Details of modified code for the if-conversion applications.	58

LIST OF FIGURES

Figure 1.1	Impact of perfect branch prediction.	2
Figure 1.2	Effect of branch mispredictions on memory latency tolerance.	3
Figure 1.3	Problems addressed by CFD.	6
Figure 2.1	Breakdown of branch mispredictions.	10
Figure 2.2	Hammock (from HMMER).	12
Figure 2.3	Totally separable branch (from SOPLEX).	13
Figure 2.4	Partially separable branch (from ASTAR).	13
Figure 2.5	Inseparable branch (from BZIP2).	14
Figure 2.6	Breakdown of targeted mispredictions.	15
Figure 3.1	High-level view of the CFD transformation for totally separable branches.	17
Figure 3.2	High-level view of the CFD transformation for partially separable branches.	18
Figure 3.3	SOPLEX source code.	21
Figure 3.4	Fields of a BQ entry.	23
Figure 3.5	BQ operation: early push (common).	24
Figure 3.6	BQ operation: late push (uncommon).	25
Figure 3.7	High-level view of the DFD transformation.	29
Figure 3.8	A generalization of the separability property.	31
Figure 3.9	ASTAR source code.	33
Figure 3.10	Fields of a TQ entry.	34
Figure 3.11	Fields of a TQ entry with overflow support.	35
Figure 3.12	Overflow-aware TQ usage.	35
Figure 3.13	Applying CFD to nested, hard-to-predict, separable branches.	37
Figure 3.14	Applying CFD to a separable branch whose control-dependent region contains a break.	40
Figure 3.15	Applying CFD to separable branches at different loop-nest levels.	42
Figure 5.1	BZIP2 source code: original and x86 vectorized.	50
Figure 5.2	BZIP2 source code vectorized for Alpha.	50
Figure 7.1	Performance and energy impact of CFD.	60
Figure 7.2	Effective IPC comparison.	61
Figure 7.3	L1 cache MHSR utilization histograms for ASTAR.	62
Figure 7.4	Varying the minimum fetch-to-execute latency.	64
Figure 7.5	CFD speedups as we scale the processor structures.	64
Figure 7.6	Effective IPC: baseline vs. CFD with/without BQ speculation.	65
Figure 7.7	Effective IPC: baseline with ISL-TAGE vs. CFD with BIMODAL, G-SHARE and ISL-TAGE.	66
Figure 7.8	ASTAR source code (region#1).	68
Figure 7.9	ASTAR: effective IPC as we scale the window size.	69
Figure 7.10	Performance and energy impact of CFD and DFD.	70

Figure 7.11	L1 cache MHSR utilization histograms for ASTAR (BigLakes, region #2).	71
Figure 7.12	Performance impact of applying CFD and DFD simultaneously.	72
Figure 7.13	Breakdown of mispredictions with respect to the furthest memory hierarchy level feeding them.	72
Figure 7.14	Effective IPC as we scale the window size: Baseline vs. CFD vs. DFD.	74
Figure 7.15	Comparison of manual and automated CFD.	75
Figure 7.16	TIFF-MEDIAN source code.	77
Figure 7.17	Breakdown of static branches in targeted applications.	77
Figure 7.18	Performance and energy impact of CFD(TQ).	78
Figure 7.19	ASTAR source code with CFD(BQ) and CFD(TQ) applied simultaneously.	79
Figure 7.20	Performance and energy impact of CFD(BQ, TQ, BQ+TQ).	79
Figure 7.21	Performance and energy impact of VEC (Alpha).	81
Figure 7.22	Wall clock time for x86 binaries.	81
Figure 7.24	Final breakdown of targeted mispredictions (MPKI) w.r.t. the proposed solutions.	82
Figure 7.23	Performance and energy impact of if-conversion.	83
Figure A.1	SPLASH-2 profiles.	105
Figure A.2	SDVBS, nBench and Firefox profiles.	105
Figure B.1	Properties of the control-flow constrained applications.	107
Figure C.1	Impact of varying the BQ size on BQ miss rate.	109
Figure C.2	Impact of varying the BQ size on speedup.	110
Figure C.3	Impact of varying the BQ size on loop strip-mining overhead.	110
Figure D.1	Design space exploration: checkpoint management.	112
Figure D.2	Design space exploration: number of checkpoints.	113
Figure D.3	Design space exploration: misprediction recovery mechanism.	113
Figure E.1	Microbenchmarks.	116
Figure E.2	The MPKIs of the unmodified microbenchmarks using three branch predictors.	117
Figure E.3	Effective IPC.	117

Chapter 1

Introduction

Good single-thread performance is important for both serial and parallel applications, and provides a degree of independence from fickle parallelism. This is why, even as the number of cores in a multi-core processor scales, processor companies continue to roll out flagship core microarchitectures that are faster than their predecessors. Meanwhile placing more cores on a chip coupled with stalled supply voltage scaling puts per-core energy consumption at a premium. Thus, the challenge is to find future microarchitecture optimizations that not only increase performance but also conserve energy.

Eliminating branch mispredictions is valuable in this respect. Mispredictions waste both time and energy, firstly, by fetching and executing wrong-path instructions and, secondly, by repairing state before resuming on the correct path. Figure 1.1a shows instructions-per-cycle (IPC) for several applications with hard-to-predict branches. The first bar is for our baseline core (refer to Table 6.2 in Chapter 6) with a state-of-art branch predictor (ISL-TAGE [55, 56]) and the second bar is for the same core with perfect branch prediction. The percentage IPC improvement with perfect branch prediction is shown above the application bars. Speedups with perfect branch prediction range from 1.05 to 2.16. Perfect branch prediction reduces energy consumption by 4% to 64% compared to real branch prediction (Figure 1.1b).

Some of these applications also suffer frequent last-level cache misses. Complexity-effective

(a) IPC

(b) Energy of perfect branch prediction relative to real branch prediction

Figure 1.1: Impact of perfect branch prediction.

large-window processors can tolerate long-latency misses and exploit memory-level parallelism with small cycle-critical structures [64, 45]. Their ability to form an effective large window is degraded, however, when a mispredicted branch depends on one of the misses [64]. Figure 1.2a shows the breakdown of mispredicted branches with respect to the furthest memory hierarchy level feeding them: L1 cache ($L1$), L2 cache ($L2$), L3 cache ($L3$) or main memory (MEM). (*NoData* represents branch mispredictions that are not memory-dependent.) The further away the memory level that feeds the branch, the longer it takes to resolve, and in the case of a misprediction, the more costly the misprediction is. Thus, as the fraction of mispredictions fed by L2, L3 and main memory increases, higher branch prediction accuracy becomes more critical to miss tolerance. This is evident in Figure 1.2b, which shows how the IPC of *astar*¹ (an application with high misprediction rate and significant fraction of mispredictions fed by L2 or L3) scales with window size. Without perfect branch prediction, IPC does not scale with window size: miss-dependent branch mispredictions prevent a large window from performing its function of latency tolerance. Conversely, eradicating mispredictions acts as a catalyst for latency tolerance. IPC scales with window size in this case.

Figure 1.2: Effect of branch mispredictions on memory latency tolerance.

¹We simulate region #1 with the reference input *Rivers*. Refer to the skip distances in Table 7.1 in Chapter 7.

We first explore the current control-flow landscape by characterizing mispredictions in four benchmark suites using a state-of-art predictor. In particular, we classify the control-dependent regions guarded by hard-to-predict branches. About a third of mispredictions-per-1K-instructions (MPKI) come from branches with small control-dependent regions, e.g., hammers. If-conversion using conditional moves, a commonly available predication primitive in commercial instruction-set architectures (ISA), is generally profitable for this class [5]. For completeness, we analyze why the gcc compiler did not if-convert such branches and manually do so at the source level in order to focus on other classes.

We discover that another third of MPKI comes from what we call *separable* branches. A separable branch has two qualities:

1. The branch has a large control-dependent region, not suitable for if-conversion.
2. The branch does not depend on its own control-dependent instructions via a loop-carried data dependence (*totally separable*), or has only a short loop-carried dependence with its control-dependent instructions (*partially separable*).

For a totally separable branch, the branch’s predicate computation is totally independent of the branch and its control-dependent region. This suggests “vectorizing” the control-flow: first generate a vector of predicates and then use this vector to drive fetching or skipping successive dynamic instances of the control-dependent region. This is the essence of our proposed technique, control-flow decoupling (CFD), for eradicating mispredictions of separable branches. The loop containing the branch is separated into two loops: a first loop contains only the instructions needed to compute the branch’s predicate (generate branch outcomes) and a second loop contains the branch and its control-dependent instructions (consume branch outcomes). The first loop communicates branch outcomes to the second loop through an architectural queue, specified in the ISA and managed by push and pop instructions. At the microarchitecture level, the queue resides in the fetch unit to facilitate timely, non-speculative branching.

Partially separable branches can also be handled. In this case, the branch’s predicate compu-

tation depends on some of its control-dependent instructions. This means a copy of the branch and the specific control-dependent instructions must be included in the first loop. Fortunately, this copy of the branch can be profitably removed by if-conversion due to few control-dependent instructions.

The novel idea of CFD targets two problems:

- Problem #1: There is insufficient *fetch separation* between the branch’s backward slice (i.e., its predicate computation) and the branch. The branch is fetched very soon after its slice, hence, it is very unlikely that the slice has executed by the time the branch is fetched. The only recourse to avoid stalling the fetch unit is to predict the branch. This problem is illustrated in Figure 1.3a, where three loop iterations are shown: iteration-a (*slice-a, branch-a*), iteration-b (*slice-b, branch-b*), and iteration-c (*slice-c, branch-c*). As shown, the branch is fetched immediately after its slice and before the slice has executed. CFD addresses this problem by separating the loop containing the branch into two loops: the first loop contains only the branch slice and the second loop contains the branch and its control-dependent instructions. This way, an instance of the branch slice is separated from its corresponding branch by other instances of the branch slice. This provides sufficient fetch separation so that the branch slice can execute before the branch is fetched (generating timely predicates). This is illustrated in Figure 1.3c (❶).
- Problem #2: Conventional processors lack support to execute branches in the fetch unit. As shown in Figure 1.3b, the slice and branch communicate through general-purpose registers (GPRs), which reside in the execution unit, so the branch must still be predicted, even if sufficient fetch separation is introduced between the branch slice and the branch. Exploiting the timely predicates, to execute the branch in the fetch stage, requires new ISA and hardware support. CFD links the slice to the branch through an architectural queue instead of GPRs, and the hardware implementation of the queue resides in the fetch unit. With fetch separation and explicit predicate communication, CFD effectively executes branches in the fetch stage. This is illustrated in Figure 1.3c (❷).

(a) Problem #1: no fetch separation between branch slice and branch

(b) Problem #2: no mechanism to execute branches in the fetch unit

(c) CFD provides the needed fetch separation (❶) and mechanism to execute branches in the fetch unit (❷)

Figure 1.3: Problems addressed by CFD.

This dissertation makes the following main contributions:

1. *CFD* [59]: A software/hardware collaboration technique that exploits branch separability with low complexity and high efficacy. The loop containing the separable branch is split into two loops (*software*): the first contains only the branch’s predicate computation and the second contains the branch and its control-dependent instructions. The first loop communicates branch outcomes to the second loop through an architectural queue (*ISA*). Microarchitecturally, the queue resides in the fetch unit to drive timely, non-speculative branching (*hardware*).
2. *CFD enhancements and derivatives*: We propose two enhancements for CFD. First, we introduce an architectural *value queue* to communicate values that are computed in the first loop, as part of the predicate computation, and needed in the second loop. Communicating the value avoids duplicating computation in the second loop, thus increasing CFD’s efficiency. Second, we introduce an architectural *trip-count queue* to communicate trip-counts to the fetch unit. This enhancement enables applying CFD to separable loop-branches as well. Furthermore, we propose data-flow decoupling (DFD), a lower-overhead derivative of CFD. Instead of eliminating mispredictions outright, DFD prefetches the misses that feed the mispredictions, thus resolving the mispredictions earlier since the loads that feed them will hit in the cache.
3. *CFD compiler pass*: We implement a compiler pass, as a plug-in for gcc, that performs the CFD transformation automatically.
4. *Analysis of inseparable branches*: We find that inseparable branches have many distinct patterns, hence, it is difficult to find one solution that fits all of them. Our analysis reveals an interesting pattern in some of the inseparable branches, whereby the branch tests contiguous memory locations (e.g., sequentially traversing an array) to locate an element. We propose vectorization (VEC) to eradicate mispredictions of this pattern.

On a microarchitecture configured similar to Intel’s Sandy Bridge core, CFD increases performance by up to 55% and reduces energy consumption by up to 49%. For hard-to-predict branches that traverse large data structures that suffer many cache misses, CFD acts as the necessary catalyst for future large-window architectures to tolerate these misses.

DFD increases performance by up to 60% and reduces energy consumption by up to 25%. On the whole, however, CFD is superior to DFD for two reasons. First, DFD only applies to a subset of the CFD-class applications. Second, as window size is increased, CFD gains scale much better than DFD gains. We conclude that attempting to speed the resolution of mispredicted branches does not compete with eliminating them altogether.

For bzip2, VEC increases performance by 37% and reduces energy consumption by 25%. For this application, the gains due to vectorization stem from two factors. First, all mispredictions of the targeted branches are eliminated, reducing the total number of mispredictions by 25%. Second, vectorization reduces dynamic instruction count by 30%.

This dissertation is organized as follows. In Chapter 2, we discuss our methodology and classification of control-flow in a wide range of applications. In Chapter 3, we present the ISA, hardware and software aspects of CFD. In Chapter 4, we describe the implementation of CFD in the gcc compiler. In Chapter 5, we discuss vectorization for inseparable branches. In Chapter 6, we describe our evaluation framework and baseline selection process. In Chapter 7, we present an evaluation of the proposed techniques. In Chapter 8, we discuss prior related work. We conclude and discuss future work in Chapter 9.

Chapter 2

Methodology and Control-Flow Classification

The goal of the control-flow classification is first and foremost discovery: to gain insight into the nature of difficult branches’ control-dependent regions, as this factor influences the solutions that will be needed, both old and new. Accordingly we cast a wide net to expose as many control-flow idioms as possible: (1) we use four benchmark suites comprised of over 80 applications, and (2) for the purposes of this comprehensive branch study, each application is run to completion leveraging a PIN-based branch profiling tool.

2.1 Methodology

We use four benchmark suites¹: *SPEC2006* [65] (engineering, scientific, and other workstation type benchmarks), *NU-MineBench-3.0* [46] (data mining), *BioBench* [4] (bioinformatics), and *cBench-1.1* [17] (embedded). All benchmarks² are compiled for x86 using gcc with optimization

¹Additional benchmark suites are profiled and found to have predictable control-flow. The details of these benchmark suites are shown in Appendix A.

²For benchmarks with multiple ref inputs, we profiled then classified all inputs into groups based on the control-flow patterns exposed. One input is selected from each group in order to cover all observed patterns. For example, for *bzip2* we select the ref inputs *input.source* and *chicken*.

level $-O3$ and run to completion using PIN [39]. We wrote a pintool that instantiates a state-of-art branch predictor (winner of CBP3, the third Championship Branch Prediction: 64KB ISL-TAGE [55]) that is used to collect detailed information for every static branch.

Different benchmarks have different dynamic instruction counts. In the misprediction contribution pie charts that follow, we weigh each benchmark equally by using its MPKI instead of its total number of mispredictions. Effectively we consider the average one-thousand-instruction interval of each benchmark.

Figure 2.1a shows the relative misprediction contributions of the four benchmark suites. Every benchmark of every suite is included³, and, as just mentioned, each benchmark is allocated a slice proportional to its MPKI. We further refine the breakdown of each benchmark suite slice into *targeted* versus *excluded*, shown in Figure 2.1b. The excluded slice contains (1) benchmarks with misprediction rates less than 2%, and (2) benchmarks that we could not run in our detailed timing simulator introduced later (due to gcc Alpha cross-compiler problems). The targeted slice contains the remaining benchmarks.

Figure 2.1: Breakdown of branch mispredictions.

³A benchmark that is present in multiple suites is included once. For example, *hmm* appears in *BioBench* and *SPEC2006*. In both benchmark suites, the same hard-to-predict branches are exposed, thus, only one instance of *hmm* is included.

This work focuses on the targeted slices which, according to Figure 2.1b, contribute almost 78% of cumulative MPKI in the four benchmark suites. Table 2.1 lists the targeted benchmarks⁴ along with their MPKIs.

Table 2.1: Targeted applications.

Benchmark Suite	Application	MPKI	Benchmark Suite	Application	MPKI
SPEC2006	astar (BigLakes)	10.11	cBench	gsm	2.10
	astar (Rivers)	25.98		jpeg-compr	8.17
	bzip2 (chicken)	4.40		jpeg-decompr	2.41
	bzip2 (input.source)	8.16		quick-sort	4.64
	gobmk	7.17		tiff-2-bw	5.42
	gromacs	1.13		tiff-median	3.60
	mcf	9.06	BioBench	clustalw	4.25
	namd	1.17		fasta	16.90
	sjeng	5.15		hmmer	12.32
	soplex (pds)	6.14	MineBench	eclat	10.19
	soplex (ref)	2.25			

2.2 Control-Flow Classification

We inspected branches in the targeted benchmarks, and categorized them into the following four classes:

1. *Hammock*: Branches with small, simple control-dependent regions. Such branches will be if-converted. From what we can tell, the gcc compiler did not if-convert these branches because they guard stores. An example from *hmmer* is shown in Figure 2.2, line 1. To encourage if-conversion, the code can be adjusted (manually or using compiler) to unconditionally perform the store, if legal (i.e., if address is legal regardless of branch outcome). The control-dependent store to $ic[k]$ (line 1) is moved outside the hammock (line 4) and the value being stored is a new local variable, *local*. Depending on the branch, *local* contains either the original value of $ic[k]$ (line 2) or *sc* (line 3). Thus, the store to $ic[k]$, after the hammock, is effectively conditional – $ic[k]$ ’s value may or may not change – even though it is performed unconditionally. The new if-statement (line 3) is then if-converted

⁴Further details about the targeted benchmarks can be found in Appendix B.

by the compiler using a conditional move (line 6): conditionally move *sc* into *local* based on the condition $sc > local$. This transformation increases the number of retired stores, but the extra stores are silent. Obtaining the original value at the memory location requires a load, but we observed that most cases are like the *hmmmer* example, in which the load already exists because the branch’s test depends on a reference to *ic[k]* (line 1).

Figure 2.2: Hammock (from HMMER).

2. *Separable*: Branches with large, complex control-dependent regions, where the branch’s backward slice (predicate computation) is either *totally separable* or *partially separable* from the branch and its control-dependent instructions.

The backward slice is *totally separable* if it does not contain any of the branch’s control-dependent instructions. An example from *soplex* is shown in Figure 2.3. The branch of interest is at line 3. (This example will be discussed in depth in Section 3.1 in Chapter 3.) Total separability allows all iterations of the backward slice to be hoisted outside the loop containing the branch, conceptually vectorizing the predicate computation. This is what control-flow decoupling does.

The backward slice is *partially separable* if it contains very few of the branch’s control-dependent instructions. An example from *astar* is shown in Figure 2.4. The branch of interest is at line 2 and the control-dependent statement in the branch’s backward slice is at line 3. In this case, the backward slice also contains the branch itself, since the branch

guards the few control-dependent instructions in the slice. All iterations of the backward slice can still be hoisted but it contains a copy of the branch, therefore, the backward slice is if-converted.

Control-flow decoupling will be applied to totally and partially separable branches.

1	for (...) {
2	x = test[i];
3	if (x < -theeps) {
4	x *= x / penalty_ptr[i];
5	x *= p[i];
6	if (x > best) {
7	best = x;
8	selId = thesolver->id(i);
9	}
10	}
11	}

Figure 2.3: Totally separable branch (from SOPLEX).

1	for (...) {
2	if (b1arp[j]->nb1ar[i]->fillnum != regfillnum) {
3	b1arp[j]->nb1ar[i]->fillnum=regfillnum;
4	b1arp[j]->nb1ar[i]->waydist=filltact;
5	flend = (b1arp[j]->nb1ar[i]==rend);
6	b2arp.add(b1arp[j]->nb1ar[i]);
7	}
8	}

Figure 2.4: Partially separable branch (from ASTAR).

3. *Inseparable*: Branches with large, complex control-dependent regions, where the branch's backward slice contains too many of the branch's control-dependent instructions. An *inseparable* branch differs from a *partially separable* branch, in that it is not profitable (or in some cases not possible) to if-convert its backward slice. This type of branch is very serial in nature: the branch is frequently mispredicted and it depends on many of the instructions that it guards. An example from *bzip2* is shown in Figure 2.5. The branch of

interest is at line 3. It guards an early return statement. This means that the execution of future branch instances depends on the branch outcome. In other words, the control-dependent region of the branch contains all future iterations, and the branch outcome depends on all previous instances of the branch. This class of branch cannot be handled by if-conversion or control-flow decoupling.

1	for (i = 0; i < 12; i++) {
2	c1 = block[i1]; c2 = block[i2];
3	if (c1 != c2) return (c1 > c2);
4	i1++; i2++;
5	}

Figure 2.5: Inseparable branch (from BZIP2).

We identify an interesting pattern in some of the inseparable branches, where the branch is testing contiguous memory locations (e.g., sequentially traversing an array) to locate an element. Figure 2.5 demonstrates this pattern. Such branches can be eliminated through vectorization.

4. *Not Analyzed*: Branches we did not analyze, i.e., branches with small contributions to total mispredictions.

Figure 2.6 breaks down the *targeted* mispredictions of Figure 2.1b into these four classes. 41.4% of the targeted mispredictions can be handled using CFD. 26.5% of the targeted mispredictions can be handled using if-conversion. That CFD covers the largest percentage of MPKI after applying a sophisticated branch predictor, provides a compelling case for its software, architecture, and microarchitecture support. Its applicability is on par with if-conversion, a commercially mainstream technique that also combines software, architecture, and microarchitecture. In addition to comparable MPKI coverage, CFD and if-conversion apply to comparable numbers of benchmarks and static branches (see Tables 7.5, 7.6 and 7.8 in Chapter 7).

Figure 2.6: Breakdown of targeted mispredictions.

Chapter 3

Control-Flow Decoupling

Figure 3.1a shows a high-level view of a totally separable branch within a loop. Branch slice (marked in *red*) computes the branch’s predicate. Depending on the predicate, the branch is taken or not-taken, causing its control-dependent instructions to be skipped or executed, respectively. In this example, none of the branch’s control-dependent instructions are in its backward slice, i.e., there isn’t a loop-carried data dependency between any of the control-dependent instructions and the branch.

Figure 3.1b shows the loop transformed for CFD. The loop is separated into two loops, each with the same trip-count as the original. The first loop has just the branch slice. It pushes predicates onto an architectural *branch queue* (BQ) using a new instruction, Push_BQ (marked in *black*). The second loop has the control-dependent instructions. They are guarded by a new instruction, Branch_on_BQ (marked in *black*). This instruction pops predicates from BQ and the predicates control whether or not the branch is taken.

Hoisting all iterations of the branch slice creates sufficient *fetch separation* between a dynamic instance of the branch and its producer instruction, ensuring that the producer executes before the branch is fetched. If successive iterations are a, b, c, ..., instead of fetching *slice-a*, *branch-a*, *slice-b*, *branch-b*, *slice-c*, *branch-c*, ..., the processor fetches *slice-a*, *slice-b*, *slice-c*, ... *branch-a*, *branch-b*, *branch-c*, Additionally, to actually exploit the now timely predicates,

Figure 3.1: High-level view of the CFD transformation for totally separable branches.

they must be communicated to the branch in the fetch stage of the pipeline so that the branch can be resolved at that time. Communicating through the existing source registers would not resolve the branch in the fetch stage. This is why we architect the BQ predicate communication medium and why, microarchitecturally, it resides in the fetch unit.

While this work assumes an OOO processor for evaluation purposes, please note that in-order and OOO processors both suffer branch penalties due to the fetch-to-execute delay of branches. We want to resolve branches in the fetch stage (so fetching is not disrupted) but they resolve in the execute stage, unless correctly predicted. Thus, the problem with branches stems from pipelining in general. OOO execution merely increases the pipeline’s speculation depth (via buffering in the scheduler) so that, far from being a solution to the branch problem, OOO execution actually makes the branch problem more acute.

Figure 3.2a shows a high-level view of a partially separable branch within a loop. A small

number of its control-dependent instructions are in the branch slice; this appears as a backward dataflow edge from these instructions to the branch slice. For a partially separable branch, the first loop would not only have (1) the branch slice and Push_BQ instruction, but also (2) the branch and just those control-dependent instructions that feed back to the branch slice, as shown in Figure 3.2b. The branch is then removed by if-conversion (marked in *green*), using conditional moves to predicate the control-dependent instructions, shown in Figure 3.2c. CFD is still profitable in this case because the subsetting control-dependent region is small and simple (otherwise the branch would be classed as inseparable).

CFD is a software-hardware collaboration. The following sections discuss ISA, software, and hardware.

Figure 3.2: High-level view of the CFD transformation for partially separable branches.

3.1 ISA Support and Benchmark Example

ISA support includes an architectural specification of the BQ and four instructions, namely: Push_BQ, Branch_on_BQ, Mark, and Forward.

The architectural specification of the BQ is as follows:

1. The BQ has a specific size. BQ size has implications for software. These are discussed in the next section.
2. Each BQ entry contains a single flag indicating taken/not-taken (the predicate). Other microarchitectural state may be included in each entry of the BQ's physical counterpart, but this state is transparent to software and not specified in the ISA.
3. A length register indicates the BQ occupancy. Architecting only a length register has the advantage of leaving low-level management concerns to the microarchitect. For example, the BQ could be implemented as a circular or shifting buffer. Thus, at the ISA level, the BQ head and tail are conceptual and are not specified as architectural registers: their physical counterparts are implementation-dependent.
4. The ISA provides mechanisms to save and restore the BQ state (queue contents and length register) to memory. This is required for context-switches. We recommend the approach used in some commercial ISAs, which is to include the BQ among the special-purpose registers and leverage move-from and move-to special-purpose-register instructions to transfer the BQ state to and from general-purpose registers (which can be saved and restored via stores and loads, respectively). If this is not possible, then dedicated Save_BQ and Restore_BQ instructions could be used.

The Push_BQ instruction has a single source register specifier to reference a general-purpose register. If the register contains zero (non-zero), Push_BQ pushes a 0 (1). Branch_on_BQ is a new conditional branch instruction. Branch_on_BQ specifies its taken-target like other conditional branches, via a PC-relative offset. It does not have any explicit source register specifiers,

however. Instead, it pops its predicate from the BQ and branches or doesn't branch, accordingly.

The ISA specifies key ordering rules for pushes and pops, that software must abide by. First, a push must precede its corresponding pop. Second, N consecutive pushes must be followed by exactly N consecutive pops in the same order as their corresponding pushes. Third, N cannot exceed the BQ size.

The Mark instruction has no register specifiers, and is used to *mark* the BQ tail entry. Similarly, the Forward instruction has no register specifiers, and is used to *bulk-pop* the BQ through to the most recently marked entry. (On a bulk-pop, the length register is decremented by the number of popped entries.) Multiple consecutive Mark instructions are allowed. A Forward instruction merely uses the last Mark. In a CFD-transformed loop, the second loop may have a smaller trip-count than the first loop, due to the original loop having an early exit condition that could not be evaluated in the first loop¹. Excess pushes from the first loop must be forcibly bulk-popped when the second loop exits early. This is achieved by inserting a Mark instruction and a Forward instruction immediately before and after the second loop, respectively. This will be demonstrated in the detailed case study of *astar* in Section 7.1.2 of Chapter 7.

Figure 3.3 shows a real example from the benchmark *soplex*. Referring to the original code: The loop compares each element of array *test[]* to variable *theeps*. The hard-to-predict branch is at line 3 and its control-dependent instructions are at lines 4-9. Neither the array nor the variable is updated inside the control-dependent region, thus, this is a totally separable branch. This branch contributes 31% of the benchmark's mispredictions (for *ref* input).

Decoupling the loop is fairly straightforward. The first loop computes predicates (lines 2-3) and pushes them onto the BQ (line 4). The second loop pops predicates from the BQ and conditionally executes the control-dependent instructions, accordingly (line 7).

An ISA enhancement must be carefully specified, so that its future obsolescence does not impede microarchitects of future generation processors. Accordingly, CFD is architected as an optional and scalable co-processor extension:

¹Typically, this scenario happens when the control-dependent region contains an early exit.

3.2 Software Side

For efficiency, the trip-counts of the first and second loops should not exceed the BQ size. This is a matter of performance, not correctness, because software can choose to spill/fill the BQ to/from memory. In practice, this is an important issue because many of the CFD-class loops iterate thousands of times whereas we specify a BQ size of 128 in this work².

We explored multiple solutions but the most straightforward one is loop strip mining. The original loop is converted to a doubly-nested loop. The inner loop is similar to the original loop but its trip-count is bounded by the BQ size. The outer loop iterates a sufficient number of times to emulate the original loop’s trip-count. Then, CFD is applied to the inner loop.

Decoupling the loop can be done either manually by the programmer or automatically by the compiler. In this work, CFD was initially applied manually which was a fairly easy task. In Chapter 4, we describe automating CFD in the gcc compiler and in Chapter 7 we evaluate how well it compares to the manual implementation.

3.3 Hardware Side

This section describes microarchitecture support for CFD. The BQ naturally resides in the instruction fetch unit. In our design, the BQ is implemented as a circular buffer with post-incremented head and tail pointers. In addition to the software-visible predicate bit, each BQ entry has the following microarchitectural state: pushed bit, popped bit, and checkpoint id. The four fields of a BQ entry are shown in Figure 3.4. Note that the pushed bit is necessary for CFD to function correctly in a pipelined processor. Meanwhile, the popped bit and checkpoint id are only needed to support speculation on a “late push” (pipelining implied), which will be defined and explained shortly.

For a correctly written program, a Push_BQ (push) instruction is guaranteed to be fetched before its corresponding Branch_on_BQ (pop) instruction. Because of pipelining, however, the

²The BQ size selection process is explained in Appendix C.

Figure 3.4: Fields of a BQ entry.

push might not execute before the pop is fetched, referred to as a late push. We explain BQ operation separately for the two possible scenarios: early push and late push.

3.3.1 Early Push

The early push scenario is depicted in Figure 3.5.

When the push instruction is fetched, it is allocated the entry at the BQ tail. It initializes its entry by clearing the pushed and popped bits. The push instruction keeps its BQ index with it as it flows down the pipeline³. When the push finally executes, it checks the popped bit in its BQ entry. It sees that the popped bit is still unset. This means the scenario is early push, i.e., the push executed before its pop counterpart was fetched. Accordingly, the push writes the predicate into its BQ entry and sets the pushed bit to signal this fact.

Later, the pop instruction is fetched. It is allocated the entry at the BQ head, which by the ISA ordering rules must be the same entry as its push counterpart. It checks the pushed bit. It sees that the pushed bit is set, therefore, it knows to use the predicate that was pushed earlier. The pop executes right away, either branching or not branching according to the predicate.

³Having the BQ index in the push instruction's payload enables it to reference its BQ entry later, when it executes OOO. This is a standard technique for managing microarchitecture FIFOs such as the reorder buffer and load and store queues.

Figure 3.5: BQ operation: early push (common).

3.3.2 Late Push

The late push scenario is depicted in Figure 3.6.

In this scenario, the pop is fetched before the push executes. As before, when the pop is fetched, it checks the pushed bit to see if the push executed. In this case the pushed bit is still unset so the pop knows that a predicate is not available. There are two options: (1) stall the fetch unit until the push executes, or (2) predict the predicate using the branch predictor. Our design implements option 2 which we call a *speculative pop*. When the speculative pop reaches the rename stage, a checkpoint is taken. (This is on top of the baseline core’s branch checkpointing policy, which we thoroughly explore in Chapter 6.) Unlike conventional branches, the speculative pop cannot confirm its prediction – this task rests with the late push instruction. Therefore, the speculative pop writes its predicted predicate and checkpoint id into its BQ entry, and signals this fact by setting the popped bit. This information will be referenced by the late push to confirm/disconfirm the prediction and initiate recovery if needed.

When the push finally executes, it notices that the popped bit is set in its BQ entry,

Figure 3.6: BQ operation: late push (uncommon).

signifying a late push. The push compares its predicate with the predicted one in the BQ entry. If they don't match, the push initiates recovery actions using the checkpoint id that was placed there by the speculative pop. Finally, the push writes the predicate into its BQ entry and sets the pushed bit.

Empirically, late pushes are very rare in our CFD-modified benchmarks, less than 0.1% of pops (one per thousand). When fully utilized by software, a 128-entry BQ separates a push and its corresponding pop by 127 intervening pushes. This typically corresponds to a push/pop separation of several hundreds of instructions, providing ample time for a push to execute before its pop counterpart is fetched.

The late push scenario can also be viewed as a *BQ miss*, because at the time of fetching the pop instruction, the predicate was not available in the BQ. In the remainder of this dissertation, we will use the term BQ miss exclusively.

3.3.3 BQ Length

The BQ length (occupancy) is the sum of two components:

1. `net_push_ctr`: This is the net difference between the number of pushes and pops retired from the core up to this point in the program's execution. The ISA push/pop ordering rules guarantee this count will always be greater than or equal to zero and less than or equal to BQ size. This counter is incremented when a push retires and decremented when a pop retires.
2. `pending_push_ctr`: This is the number of pushes in-flight in the window, i.e., the number of fetched but not yet retired pushes. It is incremented when a push is fetched, decremented when a push is retired (because it now counts against `net_push_ctr`), and possibly adjusted when a mispredicted branch resolves (see next subsection).

BQ length must be tracked in order to detect the BQ stall condition. In particular, if BQ length is equal to BQ size and the fetch unit fetches a push instruction, the fetch unit must stall. Note that the stall condition is guaranteed to pass for a bug-free program. The ISA push/pop ordering rules guarantee that there are *BQ size* in-flight pop instructions prior to the stalled push. The first one of these pops to retire will unstall the stalled push.

3.3.4 BQ Recovery

The core may need to roll back to a branch checkpoint, in the case of a mispredicted branch, or the committed state, in the case of an exception. In either case, the BQ itself needs to be repaired.

1. Preparing for misprediction recovery: Each branch checkpoint is augmented with state needed to restore the BQ to that point in the program execution. Namely, in addition to the usual checkpointed state (Rename Map Table, etc.), each checkpoint also takes a snapshot of the BQ head, tail, and mark pointers. This is a modest amount of state compared to other checkpointed state.
2. Preparing for exception recovery: Exception recovery requires maintaining committed versions of the BQ head, tail, and mark pointers, called `arch_head`, `arch_tail`, and `arch-`

mark, respectively. Arch_head and arch_tail are incremented when pops and pushes retire, respectively. Arch_mark is set to arch_tail when a Mark instruction retires. Arch_head is set to arch_mark when a Forward instruction retires.

When there is a roll-back, the BQ head, tail, and mark pointers are restored from the referenced checkpoint (on a misprediction) or their committed versions (on an exception), and all popped bits between the restored head and tail are cleared. Moreover, pending_push_ctr (the second component of BQ length) is reduced by the number of entries between the tail pointers before and after recovery (this corresponds to the number of squashed push instructions).

3.3.5 Branch Target Buffer

Like all other branch types, Branch_on_BQ is cached in the fetch unit's Branch Target Buffer (BTB) so that there is no penalty for a taken Branch_on_BQ as long as the BTB hits. The BTB's role is to detect branches and provide their taken-targets, in the same cycle that they are being fetched from the instruction cache. This information is combined with the taken/not-taken prediction (normal conditional branch) or the popped predicate (Branch_on_BQ) to select either the sequential or taken target. As with other branches, a BTB miss for a taken Branch_on_BQ results in a 1-cycle misfetch penalty (detected in next cycle).

Predicates for potential Branch_on_BQ instructions in the current fetch bundle are obtained from the BQ in parallel with the BTB access, because these predicates are always at consecutive entries starting at the BQ head.

3.4 Optimizations

This section describes two other techniques. The first is an optimization on top of CFD, called the Value Queue, that can reduce CFD instruction overheads in some cases. The second is an orthogonal application of the decoupled loops, namely, prefetching.

3.4.1 CFD+ (Value Queue)

In some CFD-transformed loops, we observed that values used to compute the predicate in the first loop are used again, thus recomputed, inside the control-dependent region in the second loop. A simple way to avoid duplication is to communicate values from the first loop to the second loop using an architectural value queue (VQ) and VQ push/pop instructions. We call this optimization CFD+.

An interesting trick to leverage existing instruction issue and register communication machinery in a superscalar core, is to map the architectural value queue onto the physical register file. This is facilitated by the *VQ renamer* in the rename stage. The VQ renamer is a circular buffer with post-incremented head and tail pointers. Its entries contain physical register mappings instead of values. The mappings indicate where the values are in the physical register file. At rename, a VQ push is allocated a destination physical register from the freelist. Its mapping is pushed at the tail of the VQ renamer. A VQ pop references the head of the VQ renamer to obtain its source physical register mapping. The queue semantics ensure the pop links to its corresponding push through its mapping. In this way, after renaming, VQ pushes and pops synchronize in the issue queue and communicate values in the execution lanes the same way as other producer-consumer pairs. The physical registers allocated to push instructions are freed when the pops that reference them retire.

The VQ renamer needs to be repaired on a misprediction or exception. We augment each checkpoint with a snapshot of the VQ head and tail pointers, and we maintain committed versions of the VQ head and tail pointers as was done for the committed versions of the BQ head and tail pointers. When there is a roll-back, the VQ head and tail pointers are restored from the referenced checkpoint (on a misprediction) or their committed versions (on an exception). Note that nothing special needs to be done to free the physical registers of squashed VQ push instructions. Existing freelist recovery takes care of freeing physical registers of all squashed register-producing instructions.

3.4.2 DFD (Prefetching)

In lieu of distancing branch-slices from branches, the first loop can be leveraged to distance load instructions from their dependents. Broadly, this is useful for overlapping misses with each other and with computation. More specifically, it can be used as a lower-overhead alternative to CFD for accelerating mispredicted branches that depend on cache misses. Whereas CFD targets the mispredictions directly, by removing them, one could instead prefetch the misses that feed the mispredictions. This doesn't eliminate mispredictions but it speeds up resolving them as the loads that feed the branch will hit in the cache.

We call the prefetching application data-flow decoupling (DFD). The first loop contains only the load affecting the branch and the load's address slice. We observed that this can lead to lower overhead in the first loop, due to no longer requiring if-converted control-dependent instructions (for partially separable branches), arbitrarily complex predicate computation, and Push_BQ instructions. Meanwhile, the second loop is the original unmodified loop.

Figure 3.7: High-level view of the DFD transformation.

Figure 3.7a shows a high-level view of a separable branch within a loop. Branch slice (marked in *red*) includes the load slice (marked in *blue*). Figure 3.7b shows the loop transformed for DFD. The first loop contains the load slice, which prefetches the data feeding the hard-to-predict branch. The second loop is the original unmodified loop.

3.5 Separable Loop-branches

Even with aggressive loop-branch predictors, some loop-branches remain hard-to-predict and contribute a noticeable fraction of mispredictions. Typically, loop-branches are hard-to-predict when they have irregular trip-counts by virtue of the trip-count being data-dependent. Moreover, these hard-to-predict loop-branches become top misprediction contributors when: (1) they have short trip-counts, and (2) they are revisited frequently, e.g., the loop is inside another loop that iterates a lot. We observed some cases where the data-dependent trip-count does not depend on the loop body, i.e., the loop-branch’s control-dependent region, therefore, the trip-count computation is separable from the loop-branch and the loop body.

Figures 3.8a and 3.8b show the generalization of separability to regular branches and loop-branches, respectively. In both cases, the separable branch/loop-branch is inside of an outer loop. Both cases have a control-dependent region, just of a different nature (forward region vs. loop body). The only distinction is predicate computation versus trip-count computation.

CFD can be applied to separable loop-branches as well. In this case, the trip-count computation is separated from the loop-branch and loop body, and the trip-counts are communicated through an architectural trip-count queue (TQ). The TQ resides in the fetch unit to drive timely, non-speculative branching. The CFD transformation for regular branches and loop-branches is shown in Figures 3.8c and 3.8d, respectively.

In this section, we highlight how CFD can be extended to support separable loop-branches.

Figure 3.8: A generalization of the separability property.

3.5.1 ISA Support and Benchmark Example

ISA support includes (1) the trip-count queue (TQ) and trip-count register (TCR), and (2) three new instructions, `Push_TQ`, `Pop_TQ`, and `Branch_on_TCR`. The TQ is similar to the BQ except that, instead of each entry containing a single predicate bit, each entry contains a single N-bit trip-count. `Push_TQ` pushes a trip-count onto the TQ. `Pop_TQ` pops a trip-count from the TQ and loads it into the TCR. `Branch_on_TCR` tests the TCR to determine whether to continue or exit the loop. If TCR is not zero, `Branch_on_TCR` decrements TCR and continues the loop. If TCR is zero, `Branch_on_TCR` exits the loop⁴.

The same push-pop ordering rules are used for the TQ as for the BQ.

Figure 3.9 shows a real example from the benchmark *astar*. Referring to the original code: the loop iterates over an array of arrays of structures. The hard-to-predict loop-branch is at line 3. Its trip-count is data-dependent and we observed it ranges from 0 to 9 (i.e., short trip-count). Although it is data-dependent, the dependence is with the outer loop and not the loop it controls, therefore, this is a separable loop-branch. This branch contributes 12% of the benchmark’s mispredictions (for *BigLakes* input). Note that the hard-to-predict loop-branch is inside the outer for loop (line 1). Decoupling the trip-count computation is fairly straightforward. The first loop computes the trip-counts and pushes them onto the TQ (line 3). The second loop pops the trip-counts from the TQ (line 7) and conditionally executes the loop instructions (lines 9 through 16), accordingly, using `Branch_on_TCR`.

3.5.2 Software Side

In this work, we specify a TQ size of 256, and we use loop strip mining when decoupling a long-running outer loop.

⁴The IBM PowerPC ISA [27] has a register similar to TCR, called the count register (CTR), designed to hold loop trip-counts. Special branch instructions (just like `Branch_on_TCR`) can decrement CTR and branch conditionally depending on whether or not CTR has reached zero. Similarly, Intel IA-64 [28] has the loop count register (LC).

	<i>Original Loop</i>
1	for (j=0; j<b1arp.elemqu; j++)
2	{
3	for (i=0; i<b1arp[j]->nb1ar.elemqu; i++) // hard-to-predict loop-branch
4	{
5	if (b1arp[j]->nb1ar[i]->fillnum!=regfillnum) {
6	b1arp[j]->nb1ar[i]->fillnum=regfillnum;
7	b1arp[j]->nb1ar[i]->waydist=filltact;
8	flend = (b1arp[j]->nb1ar[i]==rend);
9	b2arp.add(b1arp[j]->nb1ar[i]);
10	}
11	}
12	}
	<i>Decoupled Loops</i>
	<i>First Loop</i>
1	for (j=0; j<b1arp.elemqu; j++)
2	{
3	Push_TQ (b1arp[j]->nb1ar.elemqu); // push trip-count onto the TQ
4	}
	<i>Second Loop</i>
5	for (j=0; j<b1arp.elemqu; j++)
6	{
7	Pop_TQ (); // pop the trip-count
8	for (i=0; Branch_on_TCR ; i++) // predict using trip-count
9	{
10	if (b1arp[j]->nb1ar[i]->fillnum!=regfillnum) {
11	b1arp[j]->nb1ar[i]->fillnum=regfillnum;
12	b1arp[j]->nb1ar[i]->waydist=filltact;
13	flend = (b1arp[j]->nb1ar[i]==rend);
14	b2arp.add(b1arp[j]->nb1ar[i]);
15	}
16	}
17	}

Figure 3.9: ASTAR source code.

3.5.3 Hardware Side

Just like the BQ, the TQ naturally resides in the instruction fetch unit, and it is implemented as a circular buffer. In addition to the software-visible N-bit trip-count, each TQ entry has a pushed bit, shown in Figure 3.10.

The TCR also resides in the fetch unit. The TCR tracks how many iterations are left before the loop exits. TCR is loaded with a new trip-count when a Pop_TQ is fetched, and decremented when a Branch_on_TCR is fetched.

Speculating on a TQ miss is more complicated compared to speculating on a BQ miss. The complexity stems from the fact that a single TQ entry corresponds to 2^N instances of Branch_on_TCR. The overhead of maintaining misprediction recovery information, e.g., checkpoint ids

Figure 3.10: Fields of a TQ entry.

for every predicted Branch_on_TCR instruction, can be cumbersome. In our design, we opt to stall the fetch unit on a TQ miss until the Push_TQ executes.

TQ operation, length tracking, and recovery are identical to that of the BQ. Repairing the TCR in the case of branch mispredictions and exceptions requires augmenting each checkpoint with a snapshot of the TCR and maintaining a committed version of the TCR, respectively.

3.5.4 Support for Exceeding Maximum Trip-Count

If the programmer or compiler cannot guarantee that a loop’s trip-count is always less than 2^N , then the TQ cannot be used, unless we augment the ISA specification of the TQ to handle the possibility of exceeding the maximum trip-count. Accordingly, we propose the following changes to support loops that may exceed the maximum trip-count:

1. Each TQ entry is augmented with a software-visible overflow bit, as shown in Figure 3.11.
2. The Push_TQ instruction compares the trip-count being pushed to 2^N . If it is less than 2^N , it is written into the TQ and the overflow bit is cleared. If it is not less than 2^N , it is not written into the TQ and the overflow bit is set.
3. A special pop instruction is introduced, called Pop_TQ_and_Branch_on_Overflow. In addition to popping the trip-count, the new instruction specifies a target, via a PC-relative

offset, to which control is transferred if the overflow bit is set. Typically, the target is an unmodified version of the loop, as illustrated in Figure 3.12.

Figure 3.11: Fields of a TQ entry with overflow support.

Figure 3.12: Overflow-aware TQ usage.

3.6 Handling Complex Scenarios

3.6.1 Nested Hard-to-predict Separable Branches

Nested, hard-to-predict, separable branches can be handled in one of two ways: multi-level decoupling (Section 3.6.1.1) or software speculation (Section 3.6.1.2). For illustration, we consider *two* nested branches, as shown in Figure 3.13a (line 2 and line 3).

3.6.1.1 Multi-level Decoupling

Multi-level decoupling is illustrated in Figure 3.13b. The original loop is split into *three* loops. The first loop computes and pushes the first predicate (line 2). The second loop pops the first predicate (line 5); if the first predicate is true, the second predicate is computed and pushed (line 6) as the overall predicate (`true && pred2 == pred2`); if the first predicate is false, the overall predicate is false (`false && pred2 == false`), so a constant false predicate is pushed (line 8). The third loop pops the overall predicate (line 11) and uses it to fetch or skip the control-dependent region (lines 12-14).

3.6.1.2 Software Speculation

The software-speculative approach is shown in Figure 3.13c. The first loop computes both predicates, combines them, and pushes the overall predicate (line 2). Computing the second predicate, without testing the first predicate, is speculative, in that the original code would not have computed the second predicate if the first predicate were false. In general, if a computation exists only in the CFD-transformed code and it causes an exception, the exception is not valid and must be ignored. The software-speculative implementation in Figure 3.13c is *unsafe*, because: (1) computing *pred2* when *pred1* is false may generate an invalid exception, and (2) the processor will wrongly take the exception as it cannot distinguish invalid exceptions from valid ones.

Unfortunately, distinguishing between valid and invalid exceptions is not trivial. It requires

1	for (...) {
2	if (pred1) {
3	if (pred2) { // unsafe to evaluate pred2, without first checking if pred1 is true
4	... control-dependent region ...
5	}
6	}
7	}
8	
9	

(a) Original loop

	First Loop
1	for (...) {
2	Push_BQ(pred1); // push the first predicate
3	}
	Second Loop
4	for (...) {
5	Branch_on_BQ{ // pop the first predicate
6	Push_BQ(pred2); // push the second/overall predicate
7	}
8	else Push_BQ(0); // push the first/overall predicate (i.e., false)
9	}
	Third Loop
10	for (...) {
11	Branch_on_BQ{ // pop the overall predicate
12	... control-dependent region ...
13	}
14	}
15	
16	

(b) Multi-level decoupling

	First Loop
1	for (...) {
2	Push_BQ(pred1 && pred2); // push the overall predicate
3	}
	Second Loop
4	for (...) {
5	Branch_on_BQ{ // pop the overall predicate
6	... control-dependent region ...
7	}
8	}
9	
10	

(c) Concurrent predicate computation (speculation): unsafe implementation

	First Loop		
1	SPEC_BEGIN(TARGET);	n	TARGET:
2	for (...) {	n+1	for (...) {
3	Push_BQ(pred1 && pred2); // push the overall predicate	n+2	if (pred1) {
4	}	n+3	if (pred2) {
5	SPEC_END();	n+4	... control-dependent region ...
	Second Loop	n+5	}
6	for (...) {	n+6	}
7	Branch_on_BQ{ // pop the overall predicate	n+7	}
8	... control-dependent region ...	n+8	
9	}	n+9	
10	}		
11			
12			

(d) Concurrent predicate computation (speculation): safe implementation using SPEC_BEGIN and SPEC_END

Figure 3.13: Applying CFD to nested, hard-to-predict, separable branches.

two things. First, software-speculative code must be identified as such. This way, hardware will know that the exception was raised by a software-speculative instruction. Even so, the exception may or may not be valid depending on whether or not the instruction was supposed to execute. Thus, second, we need to confirm/disconfirm the execution. One way to do that is to check against the original loop.

Accordingly, we introduce two new instructions, `SPEC_BEGIN` and `SPEC_END`, as shown in Figure 3.13d. These instructions demarcate speculative code (lines 2-4) and facilitate reverting back to the corresponding non-speculative code to confirm/disconfirm speculative exceptions (line n).

The `SPEC_BEGIN` instruction does three things. First, it sets an architectural bit, the *speculation mode bit* (SMB), marking the beginning of the speculative code. Second, it specifies a target, via a PC-relative offset, that is written into an architectural register, the *exception target register* (ETR). The ETR contains the address to which control is transferred if the speculative code generates an exception. Third, it initiates coarse-grain checkpointing of the processor's architectural state. The checkpoint provides the means to restore the processor to its state before the speculative region. The `SPEC_END` instruction clears the SMB and frees the processor checkpoint.

At the microarchitecture level, the `SPEC_BEGIN` and `SPEC_END` instructions execute in the Retire stage. When the `SPEC_BEGIN` instruction retires, the SMB is set, the ETR is written with the `SPEC_BEGIN`'s target, and the processor checkpoint is created relative to `SPEC_BEGIN`. If and when the `SPEC_END` instruction retires, the SMB is cleared and the processor checkpoint is freed.

All instructions post exceptions in their ROB entries, as usual. Suppose the retirement unit sees that the ROB-head instruction has its exception bit set. If the SMB is *not* set, then the exception is non-speculative and is handled as usual. If the SMB *is* set, then the exception is speculative. In this case, the speculative exception is discarded, the processor's state is restored from the checkpoint and the checkpoint is freed, the pipeline is flushed, the SMB is cleared,

and control is transferred to the address specified in ETR. Effectively, this discards the entire speculative region and reverts to the non-CFD version of the loop (still strip-mined). So, instead of executing a strip-mined instance of the CFD loops, we execute the corresponding strip-mined instance of the original loop. Incidentally, if the exception is regenerated in the original loop, it was a valid exception, otherwise, it was an invalid exception. One way or the other, this revelation is of no consequence to the processor.

Industry prototypes [18] and commercially available processors [48, 29, 13] that feature hardware transactional memory, provide support for coarse-grain checkpointing. The support for speculative exceptions in CFD can be built on top of that support in a straightforward way. As opposed to restoring state on aborted transactions, CFD restores state when a speculative exception is detected.

3.6.2 Breaks

Another important scenario, shown in Figure 3.14a, is a separable branch (line 2) whose control-dependent region (lines 3-5) contains a “break” (line 4). The “break” is guarded by another, mostly-false predicate (line 4); otherwise, the original loop would have a short trip-count and would not be a CFD candidate. The “break” can be handled in one of two ways, both of which are software-speculative.

1. *Include “break” in the first loop* (Figure 3.14b): The first loop computes both the separable branch’s predicate and the mostly-false predicate that guards the “break” in the original loop. The two predicates are combined into an overall predicate that guards the “break” (line 4). The conditional branch corresponding to the “break” is very predictable because the overall predicate remains mostly-false. Computing the original mostly-false predicate without first testing the separable branch’s predicate is software-speculative, however, and may generate invalid exceptions. Thus, the first loop must be protected with SPEC_BEGIN/SPEC_END (lines 1, 6, and n).
2. *Do not include “break” in the first loop* (Figure 3.14c): Using the first approach, above,

1	for (...) {
2	if (pred) { // hard-to-predict branch
3	
4	if (mostly_false_predicate) break;
5	}
6	}

(a) Original loop

1	SPEC_BEGIN(TARGET);	First Loop	n	TARGET:
2	for (...) {		n+1	for (...) {
3	Push_BQ(pred); // push the predicate		n+2	if (pred) { // hard-to-predict branch
4	if (pred && mostly_false_predicate) break;		n+3	
5	}		n+4	if (mostly_false_predicate) break;
6	SPEC_END();		n+5	}
		Second Loop	n+6	}
7	for (...) {			
8	Branch_on_BQ{ // pop the predicate			
9				
10	if (mostly_false_predicate) break;			
11	}			
12	}			

(b) Include break in the first loop

1	SPEC_BEGIN(TARGET);	First Loop	n	TARGET:
2	for (...) {		n+1	for (...) {
3	Push_BQ(pred); // push the predicate		n+2	if (pred) { // hard-to-predict branch
4	}		n+3	
5	SPEC_END();		n+4	if (mostly_false_predicate) break;
6	Mark(); // mark the BQ tail pointer		n+5	}
		Second Loop	n+6	}
7	for (...) {			
8	Branch_on_BQ{ // pop the predicate			
9				
10	if (mostly_false_predicate) break;			
11	}			
12	}			
13	Forward(); // forward the current BQ head to the mark			

(c) Don't include break in the first loop

Figure 3.14: Applying CFD to a separable branch whose control-dependent region contains a break.

may cause many invalid exceptions, causing CFD to devolve to the original loop and incur rollback penalties. An alternative is to not include the “break” in the first loop. This causes the final instance of the first loop (recall that there are many instances of the first and second loops, due to strip-mining) to over-iterate. In Section 3.1, we introduced the Mark and Forward instructions, to bulk-pop the excess pushes produced by the first loop in this case (lines 6 and 13). There is another concern, however: if any of the excess instructions causes an exception, the exception is not valid and must be ignored. Thus, the first loop must be protected with SPEC_BEGIN/SPEC_END (lines 1, 5, and n). This approach may be preferred to the other approach, because invalid exceptions can only occur in the

final instance of the first loop (when it over-iterates), reverting to the original loop only for the last strip-mined instance.

3.6.3 Support for Aggressively Reordering Loads and Stores

Transforming a loop for CFD may require scheduling some loads and/or stores in the first loop and others in the second loop, changing the interleaving of loads and stores with respect to the original loop. This is not a problem if the programmer or compiler can confirm that all memory dependencies are obeyed. If incorrect orderings cannot be ruled out, however, then either CFD cannot be applied or support is needed for dynamically detecting and recovering from incorrect orderings. One way to do this is to surround the CFD loops with `SPEC_BEGIN` and `SPEC_END` and use address signatures to detect incorrect store-load ordering [68]. Which signatures are generated, and how they are compared, tested, etc., depends on the code. A signature test posts an exception if a problem is detected. An exception posted between `SPEC_BEGIN` and `SPEC_END` causes a rollback and transfers control to a strip-mined instance of the original loop.

3.6.4 Decoupling Separable Branches at Different Loop-nest Levels

Figure 3.15a shows two separable branches at different loop-nest levels. There are at least three possible ways to simultaneously decouple separable branches at different loop-nest levels.

The first approach, illustrated in Figure 3.15b, duplicates the loop-nest structure, such that the pushes of both branches are jointly scheduled in the first replica and the pops of both branches are jointly scheduled in the second replica. The global order of pushes matches the global order of pops, therefore, we say that this approach abides by a *global FIFO policy* with respect to the two branches. As such, this approach works with a single BQ, as-is.

The second and third approaches do not duplicate the loop-nest structure. Consequently, CFD is applied to the two branches independently and hierarchically, following from the original loop-nest structure. This is illustrated in Figures 3.15c and 3.15d. Globally, a hierarchical struc-

ture abides by a last-in-first-out (LIFO) policy with respect to the two branches, even as each branch's pushes and pops locally abide by a FIFO policy. The second approach (Figure 3.15c) implements a *global LIFO policy* on top of a single BQ by saving and restoring the BQ before and after the inner loop (saving the BQ also implies clearing it for use by another branch). The third approach (Figure 3.15d) implements a global LIFO policy without saves and restores, by using multiple architectural BQs: the outer loop uses BQ1 and the inner loop uses BQ2.

```

for ( ... ) {
    if (pred1) {                // first hard-to-predict branch
        .... control-dependent region A ....
    }

    for ( ... ) {
        if (pred2) {            // second hard-to-predict branch
            .... control-dependent region B ....
        }
    }
}

```

(a) Original code

```

for ( ... ) {
    Push_BQ(pred1);             // push the first predicate

    for ( ... ) {
        Push_BQ(pred2);         // push the second predicate
    }

    for ( ... ) {
        Branch_on_BQ{           // pop the first predicate
            .... control-dependent region A ....
        }
    }

    for ( ... ) {
        Branch_on_BQ{           // pop the second predicate
            .... control-dependent region B ....
        }
    }
}

```

(b) Global FIFO policy (single BQ as-is)

```

for ( ... ) {
    Push_BQ(pred1);             // push the first predicate
}

for ( ... ) {
    Branch_on_BQ{               // pop the first predicate
        .... control-dependent region A ....
    }

    SAVE_BQ;

    for ( ... ) {
        Push_BQ(pred2);         // push the second predicate
    }

    for ( ... ) {
        Branch_on_BQ{           // pop the second predicate
            .... control-dependent region B ....
        }
    }

    RESTORE_BQ;
}

```

(c) Global LIFO policy (single BQ with save/restore)

```

for ( ... ) {
    Push_BQ1(pred1);            // push the first predicate
}

for ( ... ) {
    Branch_on_BQ1{              // pop the first predicate
        .... control-dependent region A ....
    }

    for ( ... ) {
        Push_BQ2(pred2);        // push the second predicate
    }

    for ( ... ) {
        Branch_on_BQ2{          // pop the second predicate
            .... control-dependent region B ....
        }
    }
}

```

(d) Global LIFO policy (multiple BQs)

Figure 3.15: Applying CFD to separable branches at different loop-nest levels.

While the global FIFO policy seems to be the most straightforward global policy, the global

LIFO policy may be the only feasible option for certain codes. For example, the inner loop may be inside another function that cannot be in-lined. Therefore, the pushes of the two branches cannot be jointly scheduled before their pops, requiring a global LIFO policy. Another complication that may arise, is that it may not be feasible to jointly strip-mine the inner and outer loops. Both implementations of the global LIFO policy allow independent strip-mining of the inner and outer loops.

3.6.5 Bulk-push

In some of the encountered CFD-class loops (e.g., *soplex*, as shown in Figure 3.3), consecutive elements of an array are compared with a scalar value to generate consecutive predicates. This suggests using vector compare instructions to compute consecutive predicates in bulk, then using a bulk-push instruction.

Chapter 4

CFD Compiler Implementation

We implemented a compiler pass that performs the CFD transformation automatically. Our current implementation, as described in this chapter, has the following limitations. First, it applies CFD to separable branches (i.e., it uses the BQ and VQ) but not separable loop-branches (i.e., it does not use the TQ). Second, while the pass can generate CFD code for partially separable branches, there is not yet support to if-convert the branch in the first loop. Third, the various scenarios discussed in Section 3.6 are not supported: nested separable branches (i.e., the pass does not perform multi-level decoupling) and breaks in the control-dependent region (i.e., the pass does not use Mark and Forward instructions).

Our pass takes a list of hard-to-predict branches derived from profiling or the programmer as input, and transforms the inner-loop containing the hard-to-predict branches into CFD form. For brevity, we will refer to the decoupled first and second loops created by the compiler pass as the *Producer* and *Consumer*, respectively.

Algorithm 1 shows the overall CFD compiler implementation. The CFD function takes a loop and the hard-to-predict predicates within it as input. The first steps of the algorithm are inspired by the decoupled software pipelining (DSWP) algorithm presented by Ottoni *et al.* [49]. In particular, we borrow their strategy of first constructing a full program dependence graph (PDG), and then consolidating the strongly connected components (SCCs) into single

Algorithm 1 Overall CFD algorithm.

```
1: function CFD(Loop  $l$ , Predicate  $p$ )
2:    $pdg \leftarrow \text{BuildPDG}(l)$ 
3:    $dag \leftarrow \text{ConsolidateSCCs}(pdg)$ 
4:    $\text{MarkPredicateSlices}(dag, p)$ 
5:    $\text{AssignStmtsToLoops}(dag)$ 
6:   if non-empty CFD region found in  $dag$  then
7:      $producer \leftarrow l$ 
8:      $consumer \leftarrow \text{CloneLoop}(l)$ 
9:      $\text{ConnectLoops}(producer, consumer)$ 
10:    for all control flow decoupled branches,  $b$  do
11:      Insert  $\text{Push\_BQ}(\text{Predicate}(b))$  in  $producer$  just before  $b$ 
12:      Replace  $b$  in  $consumer$  with  $\text{Branch\_on\_BQ}$ 
13:    end for
14:    for all  $r$ , def'd in  $producer$  and used in  $consumer$  do
15:      Insert push in  $producer$  at definition of  $r$ 
16:      Replace definition of  $r$  in  $consumer$  with " $r = \text{Pop\_VQ}()$ "
17:    end for
18:    Remove code from  $producer$  assigned only to  $consumer$ 
19:    Remove code from  $consumer$  assigned only to  $producer$ 
20:    Final Dead and Redundant Code Elimination
21:  end if
22: end function
```

nodes in the graph, thus creating a directed acyclic graph (Lines 2-3).

If the hard-to-predict branch forms the root of a control-dependent region which can be isolated into one or more SCCs, then the branch is separable, and we use the consolidated graph to assign nodes to the Producer and Consumer. Otherwise, if the control-dependent region is part of the same SCC as the loop's exit condition, then no decoupling is possible, and our algorithm gives up.

In Line 4, we call the `MarkPredicateSlices` subroutine which carries out the following operations. For each hard-to-predict predicate in the loop, it finds its corresponding node in the *dag*. All nodes in its forward slice (all immediate successors and those reached through a depth-first search (DFS)) are marked as belonging to the Consumer. All nodes in its backward slice and itself (all immediate predecessors and those reached through a reverse DFS) are marked as belonging to the Producer.

At this point, some of the nodes in the *dag* have been scheduled in the Producer (e.g., predicate computation), Consumer (e.g., control-dependent region) or both (e.g., any part of the loop needed by the Producer and Consumer), but many nodes may remain unscheduled. For

Algorithm 2 Assign all statements to the Producer and/or Consumer.

```
1: function ASSIGN_STMTS_TO_LOOPS(PDG dag)
2:   for all  $n \in \text{dag}$  do
3:     Mark  $n$  as NoReplicate
4:     if  $n$  has no side-effects then
5:       Mark  $n$  as MaybeReplicate
6:     end if
7:   end for
8:   for all  $n \in \text{dag}$ , in topological order do
9:     if  $n$  has not been placed in a loop then
10:      place  $n$  in the Consumer
11:    end if
12:    if  $n$  placed in Producer and  $n$  marked MaybeReplicate then
13:      if  $n$  communicates to Consumer
14:        and  $\text{EstCost}(n) > \text{CommThreshold}$  then
15:           $n$  marked NoReplicate (values will be communicated)
16:        else
17:           $n$  marked Replicate
18:        end if
19:      end if
20:    end for
21: end function
```

example, any node that is both control independent and data independent from marked nodes, will need to be assigned a loop. AssignStmtsToLoops (line 5) completes the task of scheduling.

AssignStmtsToLoops. A statement must be placed in the Producer if any dependent instruction has already been placed in the Producer. Similarly, a statement must be placed in the Consumer if any statement it depends upon has already been placed in the Consumer. These rules must always be enforced. Fortunately, some flexibility does exist and can be leveraged for optimization. For example, if a statement produces no side-effects (e.g., not a store or a function call), then we can optionally schedule it in the Producer and Consumer. This flexibility allows the compiler to choose between replicating work and communicating values depending on which is more efficient. We use a simple heuristic to drive node scheduling, as shown in Algorithm 2.

In lines 2-7, each node is initially marked as NoReplicate to reflect that it must be scheduled in either the Producer or the Consumer. Next, we figure out if the node has any side-effects (e.g., stores or function calls) which would prevent replication, and if it does not, it is marked MaybeReplicate to reflect that we can possibly schedule it in both loops.

In lines 8-20, we visit all nodes in topological order (i.e., all node predecessors are processed before the node itself). This makes it easy to reason about predecessors when assigning

unscheduled nodes to loops.

In lines 9-11, we assign nodes unscheduled by MarkPredicateSlices to loops. We prefer to place a node in the Consumer unless forced to place it in the Producer. Once we know where the node will be scheduled, we need to determine if it is better to communicate or replicate any values it produces for the Consumer. Lines 12-19 form this judgement. First, we check to make sure that the node is marked MaybeReplicate. To determine if we should replicate, we compare the estimated runtime cost (*EstCost*) of the node against a minimal threshold that determines when communication will be profitable. If the node is expensive to execute, we mark the node as NoReplicate, which means that any register it defines must be communicated to the Consumer. Otherwise, we mark the node as Replicate and it will be computed in both loops. For all of our results, we use CommThreshold=2.

Final Code Generation. If a non-empty CFD region is found (line 6 of Algorithm 1), we finalize the loops and generate the code. This process is shown in lines 7-20. First, we clone the loop (line 8) and use the original as the Producer and the clone as the Consumer. Next, we connect the loops so that the program will first execute the Producer then the Consumer. This entails redirecting the exits of the Producer to the pre-header of the Consumer. The Consumer's exits remain unchanged since they are a clone of the original loop's exits. Our implementation works on an SSA graph, so we also fix the phi-nodes at the pre-header of the Consumer and exits of the Consumer. Moreover, while connecting the loops, we perform the loop strip mining transformation described in Section 3.2 in Chapter 3. This is easily accomplished by inserting a new outer loop to surround both the Producer and Consumer, and by forcing a break from Producer/Consumer every *BQ size* pushes/pops, respectively. Early breaks or returns, that are not inside the control-dependent region, are handled by keeping a loop count in the Producer (as a temporary variable) and passing that count to the Consumer for it to use as its trip count.

Next, we insert the necessary predicate and value communication. In lines 10-13, we visit all predicates that are computed in the Producer and communicated to the Consumer. We insert a Push.BQ in the Producer and place a Branch_on.BQ in the Consumer in place of the

original branch. This will always handle the hard-to-predict predicates (fed by the programmer or profiler), but additional predicates may also be included if the partitioning algorithm places a predicate in the Producer while some of its control-dependent instructions are in the Consumer. Ideally, the partitioning algorithm should limit the frequency of this case.

In lines 14-17, we insert value communication between the Producer and Consumer. Any register that is defined in the Producer and used in the Consumer must be communicated. The push is placed at the definition in the Producer and the pop is placed at the same point (the cloned register definition) in the Consumer.

Finally, the Producer and Consumer code is cleaned up. All instructions assigned the Consumer partition are removed from the Producer and vice versa for the Consumer. Then, a final dead and redundant code elimination pass eliminates other inefficiencies, like empty basic blocks and useless control-flow paths.

Chapter 5

Vectorization for Inseparable Branches

Vectorization is a general technique that replaces many scalar operations with fewer vector operations. It can reduce instruction count, execution time, and energy consumption. We find that some of the inseparable branches have an interesting pattern, where the branch is testing contiguous memory locations (e.g., sequentially traversing an array) to locate an element. This pattern can be vectorized by performing the tests in bulk leveraging existing vector support. This way, mispredictions are eliminated.

In this work, we leverage existing vector support (in x86 and Alpha) to eliminate branch mispredictions. Figure 5.1 shows a real example from the benchmark *bzip2*. The original and x86 vectorized code are shown. Referring to the original code: The loop iterates over an array of characters, reading sequential elements starting at two random indexes and looking for the first non-matching elements. The hard-to-predict branch is at line 3. This branch contributes 25% of the benchmark’s mispredictions (for *input.source* input).

Vectorizing the loop is fairly straightforward. The vectorized code uses the x86 intrinsics for Streaming SIMD Extensions 2 (SSE2). The code loads all elements of interest (lines 1 and 2), then performs a vector byte-by-byte comparison and checks if a mismatch exists (lines 3-6). If a mismatch is found, lines 7 through 11 are executed. Two vector comparisons are used to compute the returned result (lines 7 and 8). The comparison result is extracted and scanned,

More specifically, (1) the lack of support for unaligned loads, (2) the absence of vector compare if equal operation, and (3) the fact that Alpha vectors are 8-byte wide (the loop iterates over 12 bytes), significantly reduce the efficiency of the Alpha vectorized code. Figure 5.2 shows the Alpha vectorized code.

Chapter 6

Evaluation Environment

The microarchitecture presented in Chapter 3 is faithfully modeled in a detailed execution-driven, execute-at-execute, cycle-level simulator. The simulator runs Alpha ISA binaries. Recall, in Chapter 2, we used x86 binaries to locate hard-to-predict (easy-to-predict) branches, owing to our use of PIN. Our collected data confirms that hard-to-predict (easy-to-predict) branches in x86 binaries are hard-to-predict (easy-to-predict) in Alpha binaries. The predictability is influenced far more by program structure than the ISA that it gets mapped to.

The benchmarks simulated were drawn from the *BioBench* [4], *cBench* [17], *MineBench* [46] and *SPEC2006* [65] benchmark suites. All benchmarks are compiled to the Alpha ISA using gcc with -O3 level optimization. (We built gcc from scratch using the trunk SVN repository in the gcc-4 development line.) When applied, if-conversion, CFD, DFD and VEC modify the benchmark source. The modified benchmarks are verified by compiling natively to the x86 host, running them to completion, and verifying outputs (software queues are used to emulate the CFD queues).

Energy is measured using McPAT [38], which we augmented with energy accounting for the BQ (CFD, CFD+), VQ renamer (CFD+) and TQ (CFD). Per-access energy for the BQ, VQ renamer and TQ is obtained from CACTI [47] tagless rams, and every read/write access is tracked during execution.

Table 6.1: Minimum fetch-to-execute latency in cycles.

	AMD Bobcat	ARM Cortex A15	IBM Power6	INTEL Pentium 4
Fetch-to-Execute	13	15	13	20

The parameters of our baseline core are configured as close as possible to those of Intel’s Sandy Bridge core [71]. The baseline core uses the state-of-art ISL-TAGE predictor [55]. Additionally, in an effort to find the best-performing baseline, we explored the design space of misprediction recovery policies¹, including checkpoint policies (in-order vs. OoO reclamation, with confidence estimator [30] vs. without) and number of checkpoints (from 0 to 64). We confirmed that: (1) An aggressive policy (OoO reclamation, confidence-guided checkpointing) performs best. (2) The harmonic mean IPC, across all applications of all workloads, levels off at 8 checkpoints.

The fetch-to-execute pipeline depth is a critical parameter as it factors into the branch misprediction penalty. Table 6.1 shows the minimum fetch-to-execute latency (number of cycles) for modern processors from different vendors. The latency ranges from 13 to 20 cycles [9, 35, 36, 11]. We conservatively use 10 cycles for this parameter. We also perform a sensitivity study with this parameter in Section 7.1.1.

Table 6.2 shows the baseline core configuration. The checkpoint management policy and number of checkpoints remain unchanged throughout our evaluation, even for studies that scale other window resources.

Table 6.3 shows detailed storage overhead for BQ, VQ renamer, and TQ.

¹Details of the design space exploration can be found in Appendix D.

Table 6.2: Baseline core configuration.

Branch Prediction	BP: 64KB ISL-TAGE predictor - 16 tables: 1 bimodal, 15 partially-tagged. In addition to, IUM, SC, LP. - History lengths: {0, 3, 8, 12, 17, 33, 35, 67, 97, 138, 195, 330, 517, 1193, 1741, 1930} BTB: 4K entries, 4-way set-associative RAS: 64 entries
Memory Hierarchy	Block size: 64B Victim caches: each cache has a 16-entry FA victim cache L1: split, 64KB each, 4-way set-associative, 1-cycle access latency L2: unified, private for each core, 512KB, 8-way set-associative, 20-cycle access latency - L2 stream prefetcher: 4 streams, each of depth 16 L3: unified, shared among cores, 8MB, 16-way set-associative, 40-cycle access latency Memory: 200-cycle access latency
Fetch/Issue/Retire Width	4 instr./cycle
ROB/IQ/LDQ/STQ	168/54/64/36 (modeled after Sandy Bridge)
Fetch-to-Execute Latency	10-cycle
Physical RF	236
Checkpoints	8, OoO reclamation, confidence estimator (8K entries, 4-bit resetting counter, gshare index)
CFD	• BQ: 96B (128 6-bit entries) • VQ renamer: 128B (128 8-bit entries) • TQ: 160B (256 5-bit entries)

Table 6.3: Detailed storage overheads for BQ, VQ renamer, and TQ.

Branch Queue	Queue: 96B (128 6-bit entries)
	Per checkpoint: Snapshot of head, tail, and mark pointers: 3 x 7-bit
	Committed state: Committed version of head, tail, and mark pointers: 3 x 7-bit
	Length register (pending/net): 2 x 8-bit
	Subtotal = 96B + (8 x (3 x 7-bit)) + (3 x 7-bit) + (2 x 8-bit) = 96B + 25.625B = 121.625B
Value Queue Renamer	Queue: 128B (128 8-bit entries)
	Per checkpoint: Snapshot of head and tail pointers: 2 x 7-bit
	Committed state: Committed version of head and tail pointers: 2 x 7-bit
	Subtotal = 128B + (8 x (2 x 7-bit)) + (2 x 7-bit) = 128B + 15.75B = 143.75B
Trip-count Queue	Queue: 160B (256 5-bit entries)
	Per checkpoint: Snapshot of head and tail pointers: 2 x 8-bit Snapshot of TCR: 4-bit
	Committed state: Committed version of head and tail pointers: 2 x 8-bit Committed version of TCR: 4-bit
	Length register (pending/net): 2 x 9-bit
	Subtotal = 160B + (8 x ((2 x 8-bit) + 4-bit)) + ((2 x 8-bit) + 4-bit) + (2 x 9-bit) = 160B + 24.75B = 184.75B
Overall	Total = 450.125B

Chapter 7

Results and Analysis

In this chapter, we apply and evaluate CFD, DFD, VEC and if-conversion. CFD is evaluated for separable branches in Sections 7.1.1, 7.1.2 and 7.1.4, and for separable loop-branches in Section 7.1.5. When necessary, we will distinguish between these two cases using “CFD(BQ)” and “CFD(TQ)”, respectively. DFD is evaluated in Section 7.1.3, VEC is evaluated in Section 7.2, and if-conversion is evaluated in Section 7.3.

To evaluate the impact of our work on the top contributors of branch mispredictions in the targeted applications, we identify the regions to be simulated as follows. Given the set of top mispredicting branches and the functions in which they reside, we fast-forward to the first occurrence of the first encountered function of interest, warm up for 10M retired instructions, and then simulate for a certain number of retired instructions. When simulating the unmodified binary for the baseline, we simulate 100M retired instructions. When simulating binaries modified for CFD, DFD, VEC or if-conversion, we simulate as many retired instructions as needed in order to perform the same amount of work as 100M retired instructions of the unmodified binary.

Tables 7.1, 7.2, 7.3 and 7.4 show the fast-forward (skip) distances of the applications and the overheads incurred by the modified binaries. Overhead is the factor by which retired instruction count increases (e.g., 1.5 means 1.5 times) for the same simulated region. In all cases except

CFD’s *soplex(ref)* and VEC’s *bzip2(input.source)*, the modified binaries are simulated for more than 100M retired instructions¹.

Table 7.1: CFD(BQ) and DFD application skip distances and overheads.

Application	Skip (B)	Overhead			
		CFD	CFD+	DFD	Compiler CFD+
astar(BigLakes #1)	11.61	1.86	-	1.31	-
astar(BigLakes #2)	53.99	1.10	-	1.36	-
astar(Rivers #1)	0.53	1.81	-	1.30	-
astar(Rivers #2)	164.0	1.11	-	1.34	-
bzip2(chicken)	0.11	1.02	1.01	-	-
bzip2(input.source)	0.25	1.27	1.16	-	-
eclat	7.10	1.28	1.12	-	1.14
gromacs	0.74	1.03	1.02	-	1.01
jpeg-compr	0.00	1.08	1.06	-	1.06
mcf	0.70	1.15	1.14	-	1.13
namd	2.17	1.01	-	-	1.01
soplex(pds)	9.94	1.02	1.02	1.03	1.01
soplex(ref)	49.25	0.90	-	1.03	1.07
tiff-2-bw	0.00	1.00	-	-	1.00
tiff-median	0.00	1.11	-	-	-

Table 7.2: CFD(TQ) application skip distances and overheads.

Application	Skip (B)	Overhead
astar(BigLakes)	53.99	1.05
astar(Rivers)	164.0	1.05
bzip2(chicken)	177.0	1.0
bzip2(input.source)	49.56	1.0

Speedup is calculated as: $cycles_{baseline}/cycles_{modified}$, where $cycles_{baseline}$ is the number of cycles to simulate 100M instructions of the unmodified binary and $cycles_{modified}$ is the number of cycles to simulate overhead_factor x 100M instructions of the modified binary which corresponds to the same simulated region.

¹For *bzip2(input.source)*, instruction count is reduced due to replacing many scalar operations with fewer vector operations. For *soplex(ref)*, the original loop contains many variables whose live ranges overlap, increasing pressure on architectural registers and resulting in many stack spills/fills. CFD’s two loops reduce register contention by virtue of some variables shifting exclusively to the first or second loop, eliminating most of the stack spills/fills, resulting in fewer retired instructions.

Table 7.3: VEC application skip distances and overheads.

Application	Skip (B)	Overhead
bzip2(chicken)	0.04	1.04
bzip2(input.source)	0.12	0.70

Table 7.4: If-conversion application skip distances and overheads.

Application	Skip (B)	Overhead
clustalw	0.04	1.0
fasta	0.00	1.0
gsm	0.00	1.03
hmmer	0.02	1.0
jpeg-decompr	0.00	1.0
quick-sort	0.19	1.06
sjeng	0.17	1.02

Effective IPC is calculated as: $instructions_{baseline} / cycles_{scheme}$, where $instructions_{baseline}$ is the number of retired instructions of the unmodified binary and $cycles_{scheme}$ is the number of cycles to simulate the binary of the given scheme.

Tables 7.5, 7.6, 7.7 and 7.8 show detailed information about the modified source code, most importantly: (1) the affected branches and (2) the fraction of time spent in the functions containing these branches, as found by gprof-monitored native execution².

Table 7.5: Details of modified code for the CFD(BQ) applications.

Application	File name	Function	Time spent	Loop line	Branch line	Loop strip mining	Communicate values	Promote variables
astar	Way_.cpp (region #1)	makebound2	20% (BigLakes) 47% (Rivers)	57	62-63, 79-80 96-97, 113-114 130-131, 147-148 164-165, 181-182	Y	N	Y
	RegWay_.cpp (region #2)	makebound2	33% (BigLakes) 12% (Rivers)	39	43	N	N	N
bzip2	compress.c	generateMTFValues	30% (chicken) 16% (input.source)	207	214	Y	Y	N
eclat	eclat.cc	get_intersect	46%	205	207, 211	Y	Y	N
gromacs	ns.c	ns5_core	11%	1503	1507, 1508, 1510	N	Y	N
jpeg-compr	jpegdngr.c	forward_DCT	83%	231	251	N	Y	N
	jcphuf.c	encode_mcu_AC_first		488	489			
		encode_mcu_AC_refine		662	663, 686		N	
mcf	pbeampp.c	primal_bea_mpp	39%	165	171	Y	Y	N
namd	ComputeNonbondedBase.h	ComputeNonbondedUtil	5%	397	410	Y	N	N
soplex	spxsteepr.c	selectLeaveX	5% (pds)	291	295	Y	Y	Y
		selectEnterX	17% (ref)	449	452		N	
tiff-2-lw	tif_lzw.c	LZWDecode	100%	377	411	N	N	N
tiff-median	tiffmedian.c	create_colorcell	100%	725	726	Y	N	N

²The fraction of time spent in the function(s) of interest is found using *gprof* while running the x86 binaries (compiled using gcc with -O3) to completion 3 times on an idle, freshly rebooted Sandy Bridge Processor running in single-user mode.

Table 7.6: Details of modified code for the CFD(TQ) applications.

Application	File name	Function	Time spent	Loop line	Loop-branch line	Loop strip mining	Communicate values	Promote variables
astar	RegWay...cpp	makebound2	33% (BigLakes) 12% (Rivers)	35	39	Y	N	N
bzip2	decompress.c	BZ2_decompress	17% (chicken) 15% (input.source)	474	474	N	N	N

Table 7.7: Details of modified code for the VEC applications.

Application	File name	Function	Time spent	Branch Line
bzip2	blocksort.c	mainGtU	9% (chicken) 46% (input.source)	408-452

Table 7.8: Details of modified code for the if-conversion applications.

Application	File name	Function	Time spent	Branch Line
clustalw	pairalign.c	forward_pass	98%	384, 388, 391-393
		reverse_pass		436, 440, 443-444
		diff		526-527, 529-530
				555-556, 558-559
fasta	dropnfa.c	FLOCAL_ALIGN	47%	1085-1086, 1096-1097 1099-1101, 1104
gsm	add.c	gsm_div	49%	228
	long_term.c	Calculation_of_the_LTP_parameters		152
hmmer	fast_algorithms.c	P7Viterbi	100%	135-137, 142, 147
jpeg-decompr	jdhuuff.c	HUFF_EXTEND	50%	372
	jdphuff.c			207
quick-sort	qsort_large.c	compare	43%	26
sjeng	moves.c	make	10%	1305
	search.c	remove_one		515

The binaries used in our evaluation, for both baseline and CFD, are generated using slightly tweaked source code. The compiler’s limited alias analysis prevents promoting certain variables to registers, reducing the quality of the generated code. The CFD binary is affected more than baseline binary due to its dual loops. So, we manually promote these variables to registers. Interprocedural alias analysis would enable the compiler see that promotion is possible.

7.1 Control-Flow Decoupling

7.1.1 CFD and CFD+

We manually apply then evaluate: CFD and CFD+. Figure 7.1a shows that CFD increases performance by up to 51% and 16% on average, while CFD+ increases performance by up to 51% and 17% on average³. Figure 7.1b shows that CFD reduces energy consumption by up to 43% and 19% on average, while CFD+ reduces energy consumption by up to 43% and 21% on average.

Figure 7.2 compares the effective IPC of four configurations: (1) baseline (*Base*), (2) *CFD+*, (3) baseline with perfectly predicted CFD branches (*Base + PerfectCFD*), and (4) baseline with all branches perfectly predicted (*Perfect Prediction*). Three distinct behaviors are observed:

1. *Group-1*: CFD underperforms PerfectCFD for *astar* (region #1), *eclat*, *gromacs*, *tiff-2-bw*, and *tiff-median*.
2. *Group-2*: CFD matches PerfectCFD for *jpeg-compr*, *mcf*, *namd*, and *soplex(pds)*.
3. *Group-3*: CFD outperforms PerfectCFD for *astar* (region #2), *bzip2*, and *soplex(ref)*.

For most applications in *Group-1*, CFD underperforms PerfectCFD due to its instruction overheads (shown in Table 7.1). The only exception to this rule is *tiff-2-bw*. *tiff-2-bw* is the only application where no loop decoupling was performed. Instead, the branch predicate computation was hoisted far ahead within the loop. Unfortunately, when the predicate computation depends on an L1 cache miss, we suffer a BQ miss due to insufficient fetch separation. As a result, the application suffers a relatively high BQ miss rate of 20%⁴. So, 20% of the branch instances must be predicted, yielding less-than-perfect prediction.

For *Group-2* applications, the CFD instruction overheads are tolerated.

³The time spent in the functions of interest (shown in Table 7.5) along with the presented speedups, can be used in Amdahl's law to estimate the speedup of the whole benchmark. For example, *astar(Rivers, region #1)* is sped up by 34% ($s=1.34$) in its CFD region which accounts for 47% of its original execution time ($f=0.47$); thus, we estimate 14% (1.14) speedup overall.

⁴All CFD-class applications, except *tiff-2-bw*, have a 99.9% BQ hit rate.

Figure 7.2: Effective IPC comparison.

Group-3 applications demonstrate two, very interesting side-effects of CFD:

1. CFD can reduce instruction count (compared to baseline) by reducing stack spills and fills. This is the case for *soplex(ref)*, in which the original loop contains many variables whose live ranges overlap, increasing pressure on architectural registers and resulting in many stack spills/fills. CFD's two loops reduce register contention by virtue of some variables shifting exclusively to the first or second loop, eliminating most of the stack spills/fills, resulting in fewer retired instructions.
2. CFD positively impacts memory-level parallelism (MLP) by increasing the burstiness of cache misses. By virtue of splitting the original loop into two loops (each loop is smaller than the original), CFD enables more loop iterations to be in the instruction window at any given time, which increases the likelihood of having more concurrent cache misses. Instead of spreading cache misses over N consecutive instruction windows, CFD condenses the misses over M consecutive instruction windows, where $M < N$. In other words, CFD

reduces the total number of miss clusters (from N , down to M) by increasing the number of misses in a cluster. Performance is improved because more misses are overlapped within a cluster.

(a) BigLakes #2

(b) Rivers #2

Figure 7.3: L1 cache MHSR utilization histograms for ASTAR.

To confirm this phenomenon, we studied the utilization of miss handling status registers (MHSRs) at all cache levels. This phenomenon manifests in three ways: first, an increase in the fraction of time when many MHSR entries are in use; second, a decrease in the fraction of time when very few MHSR entries are in use; third, an increase in the fraction of time when zero MHSR entries are in use. Even though this behavior is observed at all cache levels, it is much more obvious in the MHSR utilization histograms of the L1 cache. Figure 7.3 shows the L1 cache MHSR utilization histograms for *astar* (region #2), one of the applications in *Group-3*. Notice how CFD+ exhibits a strong bimodal distribution in its histogram. Compared to the other cases without decoupling, CFD+ shows a large fraction of time spent in the zero-utilization bin and high-utilization bins (10 to 32), and low fraction of time in the middle-utilization bins (1-9). This is strong evidence of fewer, denser miss clusters.

Figure 7.4 shows speedup with CFD as the minimum fetch-to-execute latency is varied from five to twenty cycles. As expected, CFD gains increase as the pipeline depth increases. The baseline IPC worsens with increasing depth, whereas CFD’s eradication of mispredicted branches makes IPC insensitive to pipeline depth. Thus, as is true with better branch prediction, CFD has the added benefit of exacting performance gains from frequency scaling (i.e., deeper pipelining).

To project the gains of CFD on future processor generations, we evaluate it under larger instruction windows. Figure 7.5 shows the projection of CFD gains on two additional configurations labeled in the graph with ROB size⁵. The average performance improvement increases to 25%.

CFD-class branches inside loops that do not iterate a lot are more likely to suffer BQ misses due to the insufficient fetch separation between pushes and pops. The CFD-class branches identified in this work are inside loops that iterate a lot, making speculation on a BQ miss less critical. We evaluate CFD with and without speculation support. Figure 7.6 shows the effective

⁵[ROB,IQ,LDQ,STQ,PRF] are as follows for the two additional configurations: [256, 82, 96, 54, 324] and [384, 122, 216, 82, 452]. Other parameters match those of the baseline, shown in Table 6.2 in Chapter 6.

Figure 7.4: Varying the minimum fetch-to-execute latency.

Figure 7.5: CFD speedups as we scale the processor structures.

IPC for three configurations: (1) Baseline (*Base*), (2) CFD with speculation support (*CFD (spec)*), and (3) CFD without speculation support (*CFD (stall)*). In all applications, except *tiff-2-bw*⁶, there is no major performance loss due to not speculating (i.e., stalling) on a BQ miss. Our expectations are confirmed.

Figure 7.6: Effective IPC: baseline vs. CFD with/without BQ speculation.

Eradicating branch mispredictions (using techniques like CFD, VEC, and if-conversion) reduces our reliance on branch prediction, in general, and on complex branch predictors, in particular. When all or most mispredictions are eliminated, we can substitute complex predictors with smaller, simpler, and more generic predictors. Benefits of smaller and simpler predictors include higher clock frequency, lower energy consumption, and easier-to-pipeline branch prediction logic (for even higher frequency). Figure 7.7 shows the effective IPC of (1) baseline with ISL-TAGE predictor, and (2) CFD with BIMODAL/G-SHARE/ISL-TAGE predictors. Three trends are evident. First, in applications where CFD eliminates all or most mispredictions (e.g.,

⁶Recall that, in *tiff-2-bw*, no loop decoupling was performed.

astar region #1, *eclat*), the effective IPC of CFD does not change when varying the underlying branch predictor. Second, for all applications, except *namd* and *tiff-median*, CFD with a simple predictor (e.g., G-SHARE) delivers the same or better performance compared to the baseline with a complex predictor (ISL-TAGE). Third, in *namd* and *tiff-median*, CFD does not eliminate all mispredictions in the simulated region. Hence, a complex predictor might still be crucial for reducing the non-CFD branch midpredictions.

Figure 7.7: Effective IPC: baseline with ISL-TAGE vs. CFD with BIMODAL, G-SHARE and ISL-TAGE.

In an effort to confirm that our CFD-class branches are indeed poor candidates for predication, we compiled all of the CFD-class applications for the ARM ISA. The ARM ISA has rich predication support. For instance, ARM supports predicated move, add, subtract, load and store instructions. We observed that the gcc-arm compiler could not predicate any of the CFD-class branches, except for *namd*, where the control-dependent region is relatively small and was successfully predicated.

7.1.2 ASTAR Case Study

One of the most interesting cases we encountered in this work is *astar*, in which CFD is applied to two regions: region #1 and region #2. While both regions pose challenges, we focus the following case study on region #1, as it exhibits more challenging aspects. Figure 7.8 shows *astar*’s original and decoupled loops, for region #1. This region has three challenging features that require special care when decoupling its loop. First, there are two nested hard-to-predict branches, with the inner predicate depending on a memory reference that is only safe if the outer predicate is true (lines 3 and 4 of original loop). Second, there is a short loop-carried dependency between the outer predicate and one of its control-dependent instructions (line 7 of original loop): this is a partially separable branch. Third, the control-dependent region contains an early return statement (line 11 of the original loop).

These challenges are handled by CFD, as follows:

1. The nested conditions are handled by decoupling the original loop into *three* loops. The first loop evaluates the outermost condition. The second loop, guarded by the outermost condition, evaluates the combined condition. The third loop guards the control-dependent instructions by the overall condition.
2. The loop-carried dependency is handled by hoisting and then if-converting the short loop-carried dependency (shown in lines 10, 13 and 14 of the second loop; line 10 is also needed to evaluate the combined predicate).
3. The return statement is handled by duplicating the condition guarding the return in the second loop and replacing the return statement with an early loop break. This workaround introduces a problem: some of the predicates eagerly pushed by the first loop will not be popped by the second loop. This problem is resolved by using the Mark and Forward instructions introduced in Section 3.1 of Chapter 3. At the end of the first loop, we mark the tail of the BQ (i.e., the entry following the last predicate pushed by the first loop). At the end of the second loop, we advance the head of the BQ to the previously marked

location using the Forward instruction, which ensures that all predicates pushed by the first loop are either popped or skipped by the end of the second loop.

	<i>Original Loop</i>
1	for (...) {
2	index1=index-yoffset-1;
3	if (waymap[index1].fillnum!=fillnum) // 8 instances of this body exist
4	if (maparp[index1]==0) { // hard-to-predict branch (outer predicate)
5	bound2p[bound2l]=index1; // hard-to-predict branch (inner predicate)
6	bound2l++;
7	waymap[index1].fillnum=fillnum; // loop-carried dependency
8	waymap[index1].num=step;
9	if (index1==endindex) { // predictable branch (almost always T)
10	flend=true;
11	return bound2l;
12	}
13	}
14	}
	<i>Decoupled Loops</i>
	<i>First Loop</i>
1	for (...) {
2	index1=index-yoffset-1;
3	pred = (waymap[index1].fillnum != fillnum); // the outer predicate is computed
4	Push_BQ(pred); // then pushed onto the BQ
5	}
6	Mark(); // mark the BQ tail pointer
	<i>Second Loop</i>
7	for (...) {
8	Branch_on_BQ { // pop the outer predicate
9	index1=index-yoffset-1;
10	output = waymap[index1].fillnum;
11	pred = (output != fillnum) & (maparp[index1] == 0); // evaluate the overall predicate
12	Push_BQ(pred); // push the overall predicate
13	CMOV (output, fillnum, pred); // conditional move
14	waymap[index1].fillnum = output; // always store
15	if(index1 == local_endindex & pred) break; // return is replaced with a break
16	}
17	else Push_BQ(0); // needed since we always pop in the 3 rd loop
18	}
19	Forward(); // forward the current BQ head to the mark
	<i>Third Loop</i>
20	for (...) {
21	Branch_on_BQ { // pop the overall predicate
22	index1=index-yoffset-1;
23	bound2p[bound2l]=index1;
24	bound2l++;
25	waymap[index1].num=step;
26	if(index1==local_endindex){
27	flend=true;
28	return bound2l;
29	}
30	}
31	}

Figure 7.8: ASTAR source code (region#1).

Due to the high percentage of branch mispredictions that are fed by the L2 cache, L3 cache, and main memory, we expect a significant increase in performance gains when we apply CFD to *astar* under large instruction windows. Figure 7.9 shows the effective IPC of the unmodified binaries (*Base*), and the *CFD* binaries, as we scale the window size. Our expectations are confirmed for CFD. For example, for the *BigLakes* input and region #2, the speedup increases from 1.51 to 1.91 when window size is increased from 168 to 640.

(a) Region #1

(b) Region #2

Figure 7.9: ASTAR: effective IPC as we scale the window size.

7.1.3 DFD

We manually apply then evaluate DFD for CFD-class applications with high L2 and L3 MPKIs. L1, L2 and L3 MPKIs of the targeted applications are shown in Figure B.1 of Appendix B. Three applications stand out in terms of misses: *astar*, *soplex* and *mcf*. Even though *mcf* has high L1, L2 and L3 MPKIs, we did not apply DFD to it because the cache misses are encountered outside the CFD region.

Figure 7.10: Performance and energy impact of CFD and DFD.

Figure 7.10 compares the performance and energy impact of CFD and DFD, for *astar* and *soplex*. Figure 7.10a shows that DFD increases performance by up to 60%. Figure 7.10b shows that DFD reduces energy consumption by up to 25%. Except for *astar*(*BigLakes*), CFD yields higher speedups than DFD, although DFD performs well. CFD is always significantly more energy-efficient than DFD. Two factors contribute to DFD’s superior performance gains in *astar*(*BigLakes*): (1) CFD suffers a significantly higher instruction overhead compared to DFD (1.86 vs. 1.31, for region #1), and (2) DFD’s first loop is even more compact than CFD’s first loop, allowing DFD to have a more aggressive MLP effect. Figure 7.11 shows the L1 cache MHSR utilization histograms for *astar* (*BigLakes*, region #2). While both CFD and DFD exhibit bimodal distributions in their histograms, DFD’s is more pronounced. Compared to CFD, DFD shows a larger fraction of time spent in the zero-utilization bin and the eleven highest utilization bins (22 to 32). This is strong evidence of fewer, denser miss clusters in DFD.

Figure 7.11: L1 cache MHSR utilization histograms for ASTAR (*BigLakes*, region #2).

Interestingly, DFD and CFD can be applied simultaneously: DFD prefetches the data needed for computing the predicates in CFD, allowing the CFD loops to execute faster. Figure 7.12 shows the performance improvement when applying DFD only, CFD only, and both.

Figure 7.12: Performance impact of applying CFD and DFD simultaneously.

Figure 7.13: Breakdown of mispredictions with respect to the furthest memory hierarchy level feeding them.

We expect DFD to replace mispredictions that depend on distant cache levels with mispredictions that depend on nearby cache levels. Figure 7.13 shows the breakdown of mispredicted branches with respect to the furthest memory hierarchy level feeding them, for both baseline and DFD. Figure 7.13 confirms that DFD moves the branches’ data closer to the core, relative to the baseline.

Figure 7.14 shows the effective IPC of the unmodified binaries (*Base*), *CFD* binaries and *DFD* binaries, as we scale the window size. Interestingly, DFD performance either does not scale with window size (*astar*) or scales at a lower rate than CFD (*soplex*): attempting to speed the resolution of mispredicted branches does not compete with eliminating them altogether.

7.1.4 Automated CFD

We present results of our CFD compiler pass for eight applications: *eclat*, *gromacs*, *jpeg-compr*, *mcf*, *namd*, *soplex* (*pds* and *ref*), and *tiff-2-bw*. *gromacs* has nested hard-to-predict branches. The automated CFD binary only decouples the outer branch because our current compiler implementation does not support multi-level decoupling (discussed in Chapter 4).

Figure 7.15 compares the performance improvements and energy savings of manual CFD+ vs. automated CFD+. The two approaches yield close results for the eight applications.

As for the remaining benchmarks:

1. *astar* (Rivers and Biglakes): This benchmark has complexity that is not yet supported by our compiler pass: (1) It has a partially separable branch. Note that all other targeted benchmarks have only totally separable branches. (2) The control-dependent instruction in the branch’s backward slice is a store, hence, if-converting the backward slice requires the transformation described in Chapter 3. (3) Finally, region #1 has two nested, separable branches (one partially separable and one totally separable). We explored this complexity, in depth, in Section 7.1.2.
2. *tiff-median* and *bzip2* (*chicken* and *input.source*): Transforming the targeted loops for CFD is non-trivial. It requires an understanding of the underlying algorithm, and relies

(a) ASTAR (BigLakes, region #1)

(b) ASTAR (Rivers, region #1)

(c) ASTAR (BigLakes, region #2)

(d) ASTAR (Rivers, region #2)

(e) SOPLEX (pds)

(f) SOPLEX (ref)

Figure 7.14: Effective IPC as we scale the window size: Baseline vs. CFD vs. DFD.

(a) Speedup

(b) Energy relative to baseline

Figure 7.15: Comparison of manual and automated CFD.

on altering it slightly. Such algorithmic changes are beyond the capabilities of our pass. As an example, Figure 7.16 shows *tiff-median*'s original and decoupled loops. Consider the original loop. Observe that the data structure (*entries*) is being traversed sequentially by induction variable i . The hard-to-predict branch (at line 2) compares two consecutive elements at indices i and $i+1$. If the element at index i is greater than the element at index $i+1$, the elements are swapped (lines 3-9). This branch can be mistakenly classed as inseparable because almost half of the instructions in the branch's control-dependent region appear to be in the branch slice (lines 3-5) due to the swap, which stores to element $i+1$ which becomes element i in the next iteration. Interestingly, understanding the underlying operation of the loop provides insights into why this branch is actually partially separable. Effectively, the loop moves the element with the largest value towards the end of the array using a series of consecutive elements' comparisons. The effect of swapping elements can be mimicked, while delaying the actual swap operations, by only computing the result of the consecutive elements' comparisons. Referring to the first loop, the local variable *output* is used to mimick the swap effect. This variable holds the largest value seen so far, allowing the predicate computation to proceed without the need to predicate the complete swap operation. This way, all predicates can be computed without suffering the overhead of predicating the control-dependent instructions in the branch slice.

In an effort to identify the fraction of static branches that are separable, we invoked our compiler pass for all branches in the targeted applications. Figure 7.17 shows the breakdown of static branches into four categories:

1. Separable branches (*Separable: CFD(BQ)*) constitute 43.0% of all static branches, on average.
2. Separable loop-branches (*Separable: CFD(TQ)*) constitute 34.1% of all static branches, on average.

	<i>Original Loop</i>
1	for (i = 0; i < n; ++i) {
2	if (ptr->entries[i][1] > ptr->entries[i+1][1]) { // hard-to-predict branch
3	tmp = ptr->entries[i][0];
4	ptr->entries[i][0] = ptr->entries[i+1][0];
5	ptr->entries[i+1][0] = tmp;
6	tmp = ptr->entries[i][1];
7	ptr->entries[i][1] = ptr->entries[i+1][1];
8	ptr->entries[i+1][1] = tmp;
9	next_n = i;
10	}
11	}
	<i>Decoupled Loops</i>
	<i>First Loop</i>
1	output = ptr->entries[0][1];
2	for (i = 0; i < n; ++i) {
3	value = ptr->entries[i+1][1];
4	pred = (output > value); // the predicate is computed
5	Push_BQ(pred); // then pushed onto the BQ
6	CMOV (output, value, pred); // conditional move
7	}
	<i>Second Loop</i>
8	for (i = 0; i < n; ++i) {
9	Branch_on_BQ ; // pop the predicate
10	tmp = ptr->entries[i][0];
11	ptr->entries[i][0] = ptr->entries[i+1][0];
12	ptr->entries[i+1][0] = tmp;
13	tmp = ptr->entries[i][1];
14	ptr->entries[i][1] = ptr->entries[i+1][1];
15	ptr->entries[i+1][1] = tmp;
16	next_n = i;
17	}
18	}

Figure 7.16: TIFF-MEDIAN source code.

Figure 7.17: Breakdown of static branches in targeted applications.

3. Inseparable branches (*Inseparable: Branch*) constitute 12.4% of all static branches, on average.
4. Inseparable loop-branches (*Inseparable: Loop-branch*) constitute 10.6% of all static branches, on average.

Thoroughly evaluating control-flow decoupling for predictable, separable branches is left as future work.

7.1.5 Trip-count Queue

We manually apply then evaluate CFD for separable loop-branches. Figure 7.18 shows that CFD(TQ) increases performance by up to 5%, and reduces energy consumption by up to 6%.

Figure 7.18: Performance and energy impact of CFD(TQ).

In Chapter 3, Figure 3.9, we showed the original and decoupled loop-branch of *astar*. After eliminating the loop-branch’s mispredictions using CFD(TQ), the branch inside the inner loop (line 10 in the second loop) stands out as the main misprediction contributor. Fortunately, this branch is separable and can be targeted with CFD(BQ). Figure 7.19 shows the *astar* source code after applying CFD(BQ) and CFD(TQ) simultaneously.

Figure 7.20a shows that CFD(BQ+TQ) increases performance by up to 55%. Figure 7.20b shows that CFD(BQ+TQ) reduces energy consumption by up to 49%. Interestingly, when both

First Loop	
1	for (j=0; j<b1arp.elemqu; j++)
2	{
3	Push_TQ (b1arp[j]->nb1ar.elemqu); // push trip-count onto the TQ
4	}
Second Loop	
5	for (j=0; j<b1arp.elemqu; j++)
6	{
7	Push_TQ (b1arp[j]->nb1ar.elemqu); // push trip-count onto the TQ
8	Pop_TQ (); // pop the trip-count
9	for (i=0; Branch_on_TCR ; i++) { // predict using trip-count
10	output = b1arp[j]->nb1ar[i]->fillnum;
11	pred = (output!=regfillnum); // the predicate is computed
12	Push_BQ (pred); // then pushed onto the BQ
13	CMOV (output, regfillnum, pred); // conditional move
14	b1arp[j]->nb1ar[i]->fillnum = output; // always store
15	}
16	}
Third Loop	
17	for (j=0; j<b1arp.elemqu; j++)
18	{
19	Pop_TQ (); // pop the trip-count
20	for (i=0; Branch_on_TCR ; i++) // predict using trip-count
21	{
22	Branch_on_BQ { // pop the predicate
23	b1arp[j]->nb1ar[i]->waydist=filltact;
24	flend = (b1arp[j]->nb1ar[i]==rend);
25	b2arp.add(b1arp[j]->nb1ar[i]);
26	}
27	}
28	}

Figure 7.19: ASTAR source code with CFD(BQ) and CFD(TQ) applied simultaneously.

Figure 7.20: Performance and energy impact of CFD(BQ, TQ, BQ+TQ).

techniques are applied, performance and energy gains are greater than the sum of the two techniques' individual gains.

Just like the experiment conducted for CFD(BQ) applications, we compiled all of the CFD(TQ) applications for the ARM ISA (rich predication support), and we confirmed that the gcc-arm compiler could not predicate any of the targeted branches.

7.2 Vectorization

In this section, we evaluate the impact of vectorization on some of the inseparable branches (in *bzip2*). The original and x86 vectorized code are shown in Figure 5.1. The Alpha vectorized code is shown in Figure 5.2.

Let us first consider the Alpha binaries. Figure 7.21 shows that VEC increases performance by up to 37%, and reduces energy consumption by up to 25%. Interestingly, VEC reduces instruction count by 30% for the *input.source* input, and increases it by 4% for the *chicken* input. Unexpectedly, VEC results in slightly lower performance and higher energy consumption for input *chicken*. Two factors contribute to this effect:

1. The limited Alpha vector support reduces the quality of the vectorized code (discussed in Chapter 5).
2. The targeted branches are not mispredicted often in the simulated region. Moreover, their misprediction contribution is only 1% for the *chicken* input, as opposed to 25% for the *input.source* input.

We evaluate VEC on x86 as well. We compiled the x86 vectorized and unmodified *bzip2* code using two compilers: gcc-4.7.2 and icc-12.1. We ran the x86 binaries on an idle, freshly rebooted Sandy Bridge processor, and we recorded the wall-clock run time using linux's time command. Figure 7.22 shows the collected data. Interestingly, no slowdowns are observed for the *chicken* input, and speedups of 11% and 7% are observed for icc-12.1 and gcc-4.7.2 for the *input.source* input, respectively.

Figure 7.21: Performance and energy impact of VEC (Alpha).

Figure 7.22: Wall clock time for x86 binaries.

We compiled *bzip2* for the ARM ISA (rich predication support), and we observed that the gcc-arm compiler could not predicate any of the targeted branches.

7.3 If-Conversion

For completeness, we manually apply if-conversion (using conditional moves) to branches with small control-dependent regions (individual and nested hammocks). Figure 7.23 shows that if-conversion increases performance by up to 76% and 23% on average, and reduces energy consumption by up to 35% and 16% on average.

Note that there is no overlap between the if-converted and control-flow decoupled applications.

We compiled these same applications for the ARM ISA (rich predication support), and we observed that the gcc-arm compiler successfully predicated all of the manually if-converted branches.

7.4 Final Misprediction Breakdown

Figure 7.24 shows the final breakdown of targeted mispredictions (MPKI) with respect to the techniques proposed in this work.

Figure 7.24: Final breakdown of targeted mispredictions (MPKI) w.r.t. the proposed solutions.

Figure 7.23: Performance and energy impact of if-conversion.

Chapter 8

Related Work

This chapter reviews static and dynamic branch prediction, branch predication, branch pre-execution, static scheduling of control-flow (e.g., branch delay slots), decoupled architectures and decoupled software pipelining. We also discuss CFD in the context of related work.

8.1 Static Branch Prediction

In static branch prediction, each static branch is assigned a fixed prediction, that is used to predict all dynamic instances of the branch. Assigning static predictions to branches can be done in various ways:

1. Heuristic-based static branch prediction [61]: Simple heuristics are used, for example, predict all branches as taken/not-taken, predict backward branches as taken and forward branches as not-taken, etc.
2. Profile-based static branch prediction [21]: Branch biases gleaned from previous runs of a program are used to assign static branch predictions.
3. Evidence-based static branch prediction [10]: Neural networks (i.e., machine learning) are used to infer the branch behavior in a new program using existing programs.

8.2 Dynamic Branch Prediction

In dynamic branch prediction, predictions are based on dynamic context, such as local and/or global branch history, path information, or program values. Since the advent of two-level adaptive branch prediction [74], various ingenious techniques for dynamic branch prediction have been proposed. In this section, we sample works that represent key classes of dynamic branch predictors.

8.2.1 Anti-aliasing Predictors

The limited branch predictor budget results in branches sharing the same predictor entries. If these competing branches happen to have opposite bias, they will flip each other’s prediction, causing more mispredictions. This problem is known as destructive aliasing or negative interference. The following branch predictors attempt to reduce destructive aliasing.

The filtering technique [12] attempts to reduce destructive aliasing by dynamically identifying and steering easy-to-predict branches to a simple predictor, thus preventing them from polluting the pattern history table of a two-level branch predictor.

The Agree predictor [63] changes the way prediction table counters are interpreted. Instead of reflecting the branch direction, the prediction counters reflect whether or not the prediction agrees or disagrees with the branch’s overall bias as captured by a prediction bit in the BTB. This allows branches with opposite, strong biases, to collegially share more entries in the prediction table.

The Bi-mode branch predictor [37] divides its prediction table into two halves. One half is trained for branches that are biased not-taken and the other for branches that are biased taken. A PC-indexed chooser tracks each branch’s bias and selects which half (or mode) will be used to make a prediction. Thus, branches with different biases are steered to different halves. This reduces negative interference (and enhances positive interference) because most entries in the first half of the table tend towards not-taken and most entries in the second half of the table tend towards taken.

In the YAGS (Yet Another Global Scheme) branch predictor [19], a chooser table tracks each static branch’s bias and provides the default prediction, accordingly. The prediction table is meant to store instances when the branch does not comply with its bias. The prediction table is split into two tagged halves, or direction caches, named the T cache and NT cache. The NT cache stores the anomalies for branches that are biased taken. The T cache stores the anomalies for branches that are biased not-taken. If the chooser signals that the branch is biased taken (not-taken), the NT cache (T cache) is indexed to check if the current instance is a special case in which the prediction does not agree with the bias. If the NT cache (T cache) misses, the chooser provides the prediction. If the NT cache (T cache) hits, it provides the prediction.

The gskew [44] and 2bc-gskew [57] predictors divide the prediction table into an odd number of tables, each of which is indexed with a different hash function, yielding different interferences. At prediction time, all tables are indexed, and majority voting is used to select the final prediction.

8.2.2 Long-history Predictors

The O-GEHL predictor (O-GEometric History Length predictor) [53] uses multiple prediction tables. Each table uses a different global history length, as determined by a geometric series. Each prediction table entry is a signed saturating counter. At prediction time, all tables are indexed, and the final prediction is the sign of the summation of all accessed prediction counters.

The TAGE predictor (TAgged GEometric history length predictor) [58] uses multiple partially-tagged prediction tables. As with O-GEHL, a geometric series determines the global history lengths of the tables. Each prediction table entry has a signed saturating counter, in addition to the partial tag. At prediction time, all tables are indexed, and the final prediction is determined by the table with the longest history that hits. The TAGE predictor has been augmented with auxiliary predictors to further improve its accuracy [54, 55, 56].

The Perceptron predictor [31] uses a single layer perceptron to learn the correlation between the current branch and previous branches (it can use local, global or path history). The cor-

relations are represented using a weight vector. A higher weight means higher correlation, and the weight’s sign indicates whether it is a positive or negative correlation. The final prediction is the sign of the dot product between the history bits and the weight vector.

Affector-based predictors inspect run-time dataflow to identify correlated branches in a long global history [66, 52]. The goal is to replace the long history with the non-consecutive global history bits that are correlated, thus reducing training time and delivering the accuracy of large predictors using smaller ones.

8.2.3 Load-based Branch Predictors

The ABC predictor (Address-Branch Correlation) [22] targets branch mispredictions that depend on data that miss in the L2 cache. It exploits the observation that key elements of traversed data structures are stable, and proposes re-predicting the branch using the address of the missing load.

The EXACT predictor (EXplicit dynamic-branch prediction with ACTive updates) [3, 1] targets all data-dependent branches by correlating on the memory address(es) feeding them. A dynamic branch is uniquely identified by its ID, a hash of the address(es) it depends on. Since the branch ID might not be available at the time the branch is fetched, a proxy ID is used. Furthermore, the EXACT predictor has mechanisms to actively update the prediction table on stores.

The EXACT-S predictor [3, 2] is a software-managed version of EXACT. Shadow code conveys key information (e.g., base addresses and strides of traversed data structures) directly to the fetch unit, so that it can generate branch IDs exactly and in a timely manner. This eliminates the need for ID prediction (i.e., proxy IDs). Indexing with actual IDs instead of proxy IDs results in higher prediction accuracy, efficient use of predictor entries, and a simpler active update mechanism. In addition, EXACT-S is a reconfigurable branch predictor. When profitable, the EXACT-S predictor allocates some of the predictor’s storage to be indexed by load addresses and actively updated by stores. Otherwise, all of the predictor’s storage is used

by the default branch prediction algorithm (history based).

8.2.4 Value-based Branch Predictors

The ARVI predictor (Available Register Value Information) [16] inspects the backward slice of each load-dependent branch, terminating at the load instructions. The live-in registers of the slice (destination registers of the loads) are noted by the predictor. When the branch is fetched, the predictor uses the live-in register values of its slice, if they have been committed, to predict the branch’s direction.

BPVP (Branch Prediction through Value Prediction) [23] predicts the branch’s source register values to calculate its direction early in the pipeline.

The BDP predictor (Branch Difference Predictor) [24] predicts a branch using (1) the difference between the two source operands of a prior committed instance of the same static branch, and (2) the number of outstanding instances of the same static branch in the processor window.

8.2.5 Loop-branch Predictors

The LTP predictor (Loop Termination Prediction) [60] targets loop-branch mispredictions, by capturing the loop’s repeating trip-count. It predicts that a loop will iterate as many times as it did the last time.

Many branch predictors have been augmented with loop predictors to further improve their accuracy (e.g., L-TAGE [54]). Just like LTP, these loop predictors capture repeating trip-counts only.

CFD’s trip-count queue (TQ) is innovative with respect to the above, in several ways: (1) CFD targets loop-branches that are data-dependent, i.e., CFD captures irregular, data-dependent trip-counts. (2) CFD always delivers correct trip-counts.

A fully-featured EXACT predictor [3, 1], discussed in Section 8.2.3, can also successfully predict data-dependent loop-branches. To predict data-dependent loop-branches, EXACT includes a loop predictor indexed by branch ID and actively updated by stores. Unlike CFD’s

TQ, however, the trip-count predictions are not always correct. Moreover, overall, CFD is much less resource intensive than EXACT and branch predictors in general.

8.3 Branch Predication

Predication is a form of conditional execution that does not use branches. It was originally proposed in the context of automatic vectorization [67, 5]. In a code transformation called *if-conversion*, the branch is removed and its control-dependent instructions are each guarded by the branch’s predicate.

In this work, if-conversion is a key enabling technique for applying CFD to partially separable branches. For ISA support, we used conditional move instructions, instead of predicate registers and predicated instructions.

Hyperblock scheduling [41] exploits predicated execution to fuse multiple frequently-executed paths within a single scheduling region. Hyperblocks increase the compiler’s instruction scheduling scope, exposing more instruction-level parallelism.

The wish branch technique [33] is a hybrid between branch prediction and branch predication. The compiler predicates the wish branch’s control-dependent instructions but also keeps the branch. At run-time, a confidence estimator decides whether to predict the branch (high confidence) or apply predicated execution (low confidence).

Dynamic hammock predication [34] is an all-hardware technique for replacing simple control-flow (unnested hammocks) with predicated execution. The Diverge-Merge Processor (DMP) [32] is a mostly-hardware approach to predicated execution. Like dynamic hammock predication [34], it does not require predicate registers or predicated instructions in the ISA. It achieves higher coverage of branch mispredictions, however, by exploiting the empirical observation that complex nested control-flow regions appear as simple hammocks when only frequently-executed paths are considered.

8.4 Branch Pre-execution

CFD resembles branch pre-execution [20, 75, 51, 14, 15]. The key difference is that CFD preserves the simple sequencing model of conventional superscalar processors: in-order instruction fetching of a single thread. This is in contrast with pre-execution which requires thread contexts or cores, and a suite of mechanisms for forking helper threads (careful timing, value prediction, etc.) and coordinating them in relation to the main thread. With CFD, a simplified microarchitecture stems from software/hardware collaboration, simple ISA push/pop rules, and recognition that multiple threads are not required for decoupling.

We now discuss several branch pre-execution solutions in more detail.

8.4.1 Pre-execution Using Helper Threads

Farcy *et. al.* [20] identified backward slices of applicable branches, and used a stride value predictor to provide live-in values to the slices and in this way compute predictions several loop iterations in advance. The technique requires a value predictor and relies on live-in value predictability. CFD does not require either.

Zilles and Sohi [75] proposed pre-executing backward slices of hard-to-predict branches and frequently-missed loads using *speculative slices*. Fork point selection, construction and speculative optimization of slices were done manually. Complex mechanisms are needed to carefully align branch predictions generated by speculative slices with the correct dynamic branch instances. Meanwhile, CFD’s push/pop alignment is far simpler, always delivers correct predicates, and has been automated in the compiler.

Roth and Sohi [51] developed a profile-driven compiler to extract data-driven threads (DDTs) to reduce branch and load penalties. The threads are non-speculative and their produced values can be integrated into the main thread via register integration. Branches execute more quickly as a result. Similarly, CFD is non-speculative and automation is demonstrated in this work. CFD interacts directly with the fetch unit, eliminating the entire branch penalty. It also does not have the microarchitectural complexity of register integration. The closest as-

pect is the VQ renamer, but the queue-based linking of pushes and pops via physical register mappings is simpler, moreover, it is an optional enhancement for CFD.

Chappell *et al.* [14] proposed Simultaneous Subordinate Microthreading (SSMT) as a general approach for leveraging unused execution capacity to aid the main thread. Originally, programmer-crafted subordinate microthreads were used to implement a large, virtualized two-level branch predictor. Subsequently, an automatic run-time microthread construction mechanism was proposed for pre-executing branches [15].

8.4.2 Pre-execution Leveraging Predicate Register File

Mahlke and Natarajan [40] implemented a predicate register file in the fetch stage, a critical advance in facilitating software management of the fetch unit of pipelined processors. The focus of the work, however, was compiler-synthesized branch prediction: synthesizing computation to generate predictions, writing these predictions into the fetch unit’s predicate register file, and then having branches reference the predicate registers as *predictions*. The synthesized computation correlates on older register values because the branch’s source values are not available by the time the branch is fetched, hence, this is a form of branch prediction. Mahlke *et al.* alluded to the theoretical possibility of truly resolving branches in the fetch unit, and August *et al.* [6] further explored opportunities for such *early-resolved branches*: cases where the existing predicate computation is hoisted early enough for the consuming branch to resolve in the fetch unit. These cases tend to exist in heavily if-converted code such as hyperblocks as these large scheduling regions yield more flexibility for code motion. Quinones *et al.* [50] adapted the predicate register file for an OOO processor, and in so doing resorted to moving it into the rename stage so that it can be renamed. Thus, the renamed predicate register file serves as an overriding branch predictor for the branch predictor in the fetch unit. CFD is innovative with respect to the above, in several ways: (1) The BQ provides renaming implicitly by allocating new entries at the tail. This allows for hoisting all iterations of a branch’s backward slice ahead of the loop, whereas it is unclear how this can be done with an indexed predicate register file

as the index is static. (2) Another advantage is accessing the BTB (to detect Branch_on_BQ instructions) and BQ in parallel, because we always examine the head of the queue. In contrast, accessing a predicate register file requires accessing the BTB first, to get the branch’s register index, and then accessing the predicate register file.

8.5 Static Scheduling

NSR [8] does not predict branches at all, rather, a branch waits in the fetch stage for an enqueued outcome from the execute stage. To avoid fetch stalls, a few instructions must be scheduled by the programmer or compiler in between the branch and its producer instruction. This is like branch delay slots [25] except that, because the fetch unit can stall, no explicit NOPs need to be inserted when no useful instructions can be scheduled. NSR is a 5-stage in-order pipeline so its static scheduling requirement is of similar complexity to branch delay slot scheduling. Meanwhile, CFD-class branches require our “deep” static scheduling technique (for in-order and out-of-order pipelines, alike) which in turn requires CFD’s ISA, software, and hardware support.

8.6 Decoupled Architectures

Decoupled access/execute architectures [62, 7] are alternative implementations of OOO execution, and not a technique for hiding the fetch-to-execute penalty of mispredicted branches. DAE’s access and execute streams, which execute on dual cores, each have a subset of the original program’s branches. To keep them in sync on the same overall control-flow path, they communicate branch outcomes to each other through queues. However, each core still suffers branch penalties for its subset of branches. Bird *et al.* took DAE a step further and introduced a third core for executing all control-flow instructions, the control processor (CP). CP directs instruction fetching for the other two cores (AP and DP). CP depends on branch conditions calculated in the DP, however. These loss-of-decoupling (LOD) events are equivalent to exposing

the fetch-to-execute branch penalty in a modern superscalar processor.

In the Branch Decoupled Architecture (BDA), proposed by Tyagi *et al.* [69], the fetch unit steers copies of the branch slice to a dedicated core as the unmodified dynamic instruction stream is fetched. Creating the pre-execution slice as main thread instructions are being fetched provides no additional fetch separation between the branch’s backward slice and the branch, conflicting with more recent evidence of the need to trigger helper threads further in advance, e.g., Zilles and Sohi [75]. Without fetch separation, the branch must still be predicted and its resolution may be marginally accelerated by a dedicated execution backend for the slice.

8.7 Decoupled Software Pipelining

The concept of loop decoupling has been applied in compilers for parallelization. For instance, decoupled software pipelining [49, 70, 26] parallelizes a loop by creating decoupled copies of the loop on two or more cores that cooperate to execute each iteration. All predicates in the backward slices of instructions in the decoupled loops that are not replicated must be communicated. However, predicates are not sent directly to the instruction fetch unit of the other core. Rather, the predicates are forwarded as values through memory or high speed hardware queues and evaluated in the execution stage by a branch instruction.

Chapter 9

Conclusion and Future Work

9.1 Conclusion

In this work, we explored the control-flow landscape by characterizing branches with high misprediction contributions in four benchmark suites. We classified branches based on the sizes of their control-dependent regions and the nature of their backward slices (predicate computation), as these two factors give insight into possible solutions. This exercise uncovered an important class of high misprediction contributors, called separable branches. A separable branch has a large control-dependent region, too large for if-conversion to be profitable, and its backward slice does not contain any of the branch’s control-dependent instructions or contains just a few. This makes it possible to separate all iterations of the backward slice from all iterations of the branch and its control-dependent region. CFD is a software/hardware collaboration for exploiting separability with low complexity and high efficacy. The loop containing the separable branch is split into two loops (*software*): the first contains only the branch’s predicate computation and the second contains the branch and its control-dependent instructions. The first loop communicates branch outcomes to the second loop through an architectural queue (*ISA*). Microarchitecturally, the queue resides in the fetch unit to drive timely, non-speculative fetching or skipping of successive dynamic instances of the control-dependent region (*hardware*).

Measurements of native execution of four benchmark suites show separable branches are an important class of branches, comparable to the class of branches for which if-conversion is profitable both in terms of number of static branches and MPKI contribution. CFD eradicates mispredictions of separable branches, yielding significant time and energy savings for regions containing them.

9.2 Future Work

Several directions can be explored as future work:

1. *Apply CFD to more branches:* We showed in Section 7.1.4 (Figure 7.17) that 77% of the static branches in the targeted applications are separable. Targeting branches that are mispredicted infrequently might improve performance and energy in at least two respects. First, it reduces pressure on branch checkpoints because control-flow decoupling circumvents branch prediction. Reducing checkpoint pressure leads to larger instruction windows for the same number of checkpoints (higher IPC) or reduces the number of checkpoints required for the same window size (higher frequency and lower energy consumption). Second, targeting branches with intricate patterns, that can be accurately predicted by large, complex, and specialized predictors, may enable substituting such predictors with smaller, simpler, and more generic predictors. That is, handling intricate branches with control-flow decoupling might make complex predictors redundant. Benefits of smaller and simpler predictors include higher clock frequency, lower energy consumption, and easier-to-pipeline branch prediction logic (for even higher frequency). A microbenchmark that demonstrates this point is shown in Appendix E.
2. *Improve compiler implementation:* Our current compiler pass can be extended to support:
 - (a) Partially separable branches.
 - (b) Multi-level decoupling (i.e., nested hard-to-predict branches).
 - (c) Control-dependent regions containing early break conditions (i.e., support Mark and Forward instructions).
 - (d) Separable loop-branches.

3. *Investigate solutions for inseparable branches:* Figure 7.24 shows that inseparable branches account for 14.6% of the targeted MPKI. An inseparable branch and its control-dependent region constitute a very serial program fragment: the branch is frequently mispredicted and its source operands depend on nearly all of its control-dependent instructions. This scenario calls for better branch prediction – to avoid squashing of control-independent instructions after the “rough patch” – and/or very fast serial execution – to more quickly get past the “rough patch”. For instance, we can accelerate the execution of the serial code using an alternate core that is very streamlined for serial code, i.e., a narrow, shallow pipeline. The main core waits while the accelerator core executes the serial code. Therefore, strictly speaking, this is not about eliminating branch mispredictions. Rather, it is about executing the serial code faster than on the main core which has the advantage of a complex branch predictor but the drawback of a rather large fetch-to-execute logic delay.

REFERENCES

- [1] Muawya Al-Otoom, Elliott Forbes, and Eric Rotenberg. EXACT: Explicit Dynamic-Branch Prediction with Active Updates. In *7th ACM Int'l Conf. on Computing Frontiers*, pages 165–176, 2010.
- [2] Muawya Al-Otoom, Rami Sheikh, and Eric Rotenberg. A Case for a Software-Managed Reconfigurable Branch Predictor. In *5th Workshop on Architectural and Microarchitectural Support for Binary Translation*, 2012.
- [3] Muawya M. Al-Otoom. *EXACT: Explicit Dynamic-Branch Prediction with Active Updates*. PhD thesis, Department of Electrical and Computer Engineering, North Carolina State University, 2010.
- [4] Kursad Albayraktaroglu, Aamer Jaleel, Xue Wu, Manoj Franklin, Bruce Jacob, Chau-Wen Tseng, and Donald Yeung. BioBench: A Benchmark Suite of Bioinformatics Applications. In *Int'l Symp. on Performance Analysis of Systems and Software*, pages 182–188, 2005.
- [5] John Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of Control Dependence to Data Dependence. In *10th Symp. on Principles of Programming Languages*, pages 177–189, 1983.
- [6] David I. August, Daniel A. Connors, John C. Gyllenhaal, and Wen-mei W. Hwu. Architectural Support for Compiler-synthesized Dynamic Branch Prediction Strategies: Rationale and Initial Results. In *3rd Int'l Symp. on High-Performance Computer Architecture*, pages 84–93, 1997.
- [7] Peter L. Bird, Alasdair Rawsthorne, and Nigel P. Topham. The Effectiveness of Decoupling. In *7th Int'l Conf. on Supercomputing*, pages 47–56, 1993.
- [8] Erik Brunvand. The NSR Processor. In *26th Hawaii Int'l Conf. on System Sciences*, volume 1, pages 428–435, 1993.
- [9] Brad Burgess, Brad Cohen, Marvin Denman, Jim Dundas, David Kaplan, and Jeff Rupley. Bobcat: AMD's Low-power x86 Processor. *IEEE Micro*, 31(2):16–25, 2011.
- [10] Brad Calder, Dirk Grunwald, Donald Lindsay, James Martin, Michael Mozer, and Benjamin Zorn. Corpus-based Static Branch Prediction. In *ACM SIGPLAN 1995 Conf. on Programming Language Design and Implementation*, pages 79–92, 1995.
- [11] Doug Carmean. Inside the Pentium 4 Processor Micro-Architecture. Presented at Intel Developer Forum, 2000.
- [12] Po-Yung Chang, Marius Evers, and Yale N. Patt. Improving Branch Prediction Accuracy by Reducing Pattern History Table Interference. In *1996 Conf. on Parallel Architectures and Compilation Techniques*, pages 48–57, 1996.

- [13] Robert Chappell, Bret Toll, and Ronak Singhal. Intel Next Generation Microarchitecture Codename Haswell: New Processor Innovations. Presented at Intel Developer Forum, San Francisco, 2012.
- [14] Robert S. Chappell, Jared Stark, Sangwook P. Kim, Steven K. Reinhardt, and Yale N. Patt. Simultaneous Subordinate Microthreading (SSMT). In *26th Int'l Symp. on Computer Architecture*, pages 186–195, 1999.
- [15] Robert S. Chappell, Francis Tseng, Adi Yoaz, and Yale N. Patt. Difficult-path Branch Prediction using Subordinate Microthreads. In *29th Int'l Symp. on Computer Architecture*, pages 307–317, 2002.
- [16] Lei Chen, Steve Dropsho, and David H. Albonesi. Dynamic Data Dependence Tracking and its Application to Branch Prediction. In *9th Int'l Symp. on High-Performance Computer Architecture*, pages 65–76, 2003.
- [17] cTuning. Collective Benchmark. In <http://cTuning.org/cbench>.
- [18] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early Experience with a Commercial Hardware Transactional Memory Implementation. In *14th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 157–168, 2009.
- [19] Avinoam Nomik Eden and Trevor Mudge. The YAGS Branch Prediction Scheme. In *31st Int'l Symp. on Microarchitecture*, pages 69–77, 1998.
- [20] Alexandre Farcy, Olivier Temam, Roger Espasa, and Toni Juan. Dataflow Analysis of Branch Mispredictions and its Application to Early Resolution of Branch Outcomes. In *31st Int'l Symp. on Microarchitecture*, pages 59–68, 1998.
- [21] Joseph A. Fisher and Stefan M. Freudenberger. Predicting Conditional Branch Directions From Previous Runs of a Program. In *5th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 85–95, 1992.
- [22] Hongliang Gao, Yi Ma, Martin Dimitrov, and Huiyang Zhou. Address-branch Correlation: A Novel Locality for Long-latency Hard-to-predict Branches. In *14th Int'l Symp. on High Performance Computer Architecture*, pages 74 –85, 2008.
- [23] Jose Gonzalez and Antonio Gonzalez. Control-Flow Speculation Through Value Prediction for Superscalar Processors. In *1999 Int'l Conf. on Parallel Architectures and Compilation Techniques*, pages 57–65, 1999.
- [24] Timothy H. Heil, Zak Smith, and James E. Smith. Improving Branch Predictors by Correlating On Data Values. In *32nd Int'l Symp. on Microarchitecture*, pages 28–37, 1999.
- [25] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 2003.

- [26] Jialu Huang, Arun Raman, Thomas B. Jablin, Yun Zhang, Tzu-Han Hung, and David I. August. Decoupled Software Pipelining Creates Parallelization Opportunities. In *8th Int'l Symp. on Code Generation and Optimization*, pages 121–130, 2010.
- [27] IBM. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. Morgan Kaufmann Publishers Inc., 1994.
- [28] Intel. *IA-64 Application Developers Architecture Guide*. Intel Corporation, Order Number 245188-001, 1999.
- [29] Christian Jacobi, Timothy Slegel, and Dan Greiner. Transactional Memory Architecture and Implementation for IBM System Z. In *45th Int'l Symp. on Microarchitecture*, pages 25–36, 2012.
- [30] Erik Jacobsen, Eric Rotenberg, and James E. Smith. Assigning Confidence to Conditional Branch Predictions. In *29th Int'l Symp. on Microarchitecture*, pages 142–152, 1996.
- [31] Daniel A. Jiménez and Calvin Lin. Dynamic Branch Prediction with Perceptrons. In *7th Int'l Symp. on High-Performance Computer Architecture*, pages 197–206, 2001.
- [32] Hyesoon Kim, Jose A. Joao, Onur Mutlu, and Yale N. Patt. Diverge-Merge Processor (DMP): Dynamic Predicated Execution of Complex Control-flow Graphs Based on Frequently Executed Paths. In *39th Int'l Symp. on Microarchitecture*, pages 53–64, 2006.
- [33] Hyesoon Kim, Onur Mutlu, Jared Stark, and Yale N. Patt. Wish Branches: Combining Conditional Branching and Predication for Adaptive Predicated Execution. In *38th Int'l Symp. on Microarchitecture*, pages 43–54, 2005.
- [34] Artur Klauser, Todd Austin, Dirk Grunwald, and Brad Calder. Dynamic Hammock Predication for Non-predicated Instruction Set Architectures. In *1998 Int'l Conf. on Parallel Architectures and Compilation Techniques*, pages 278–285, 1998.
- [35] Travis Lanier. Exploring the Design of the Cortex-A15 Processor. Presented at ARM Techcon, Santa Clara, CA, 2011.
- [36] Hung Q. Le, William J. Starke, Stephen Fields, Francis P. OConnell, Dung Q. Nguyen, Bruce J. Ronchetti, Wolfram M. Sauer, Eric M. Schwarz, and Michael T. Vaden. IBM POWER6 Microarchitecture. *IBM Journal of Research and Development*, 51(6):639–662, 2007.
- [37] Chih-Chieh Lee, I-Cheng K. Chen, and Trevor N. Mudge. The Bi-mode Branch Predictor. In *30th Int'l Symp. on Microarchitecture*, pages 4–13, 1997.
- [38] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *42nd Int'l Symp. on Microarchitecture*, pages 469–480, 2009.

- [39] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 190–200, 2005.
- [40] Scott Mahlke and Balas Natarajan. Compiler Synthesized Dynamic Branch Prediction. In *29th Int’l Symp. on Microarchitecture*, pages 153–164, 1996.
- [41] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective Compiler Support for Predicated Execution Using The Hyperblock. In *25th Int’l Symp. on Microarchitecture*, pages 45–54, 1992.
- [42] Uwe F. Mayer. The BYTEmark Benchmark Program, 1996.
- [43] Scott McFarling. Combining Branch Predictors. Technical Report 36, Digital Western Research Laboratory, 1993.
- [44] Pierre Michaud, André Seznec, and Richard Uhlig. Trading Conflict and Capacity Aliasing in Conditional Branch Predictors. In *24th Int’l Symp. on Computer Architecture*, pages 292–303, 1997.
- [45] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors. In *9th Int’l Symp. on High-Performance Computer Architecture*, pages 129–140, 2003.
- [46] Ramanathan Narayanan, Berkin Ozisikyilmaz, Joseph Zambreno, Gokhan Memik, and Alok Choudhary. MineBench: A Benchmark Suite for Data Mining Workloads. In *Int’l Symp. on Workload Characterization*, pages 182–188, 2006.
- [47] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P. Jouppi. CACTI 6.0: A Tool to Model Large Caches. Technical Report 85, HP Laboratories, 2009.
- [48] Martin Ohmacht, Amy Wang, Thomas Gooding, Ben Nathanson, Indira Nair, Geert Janssen, Marcel Schaal, and Burkhard Steinmacher-Burow. IBM Blue Gene/Q Memory Subsystem with Speculative Execution and Transactional Memory. *IBM Journal of Research and Development*, 57(1/2):7:1–7:12, 2013.
- [49] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. Automatic Thread Extraction with Decoupled Software Pipelining. In *38th Int’l Symp. on Microarchitecture*, pages 105–118, 2005.
- [50] Eduardo Quinones, Joan-Manuel Parcerisa, and Antonio Gonzaleiz. Improving Branch Prediction and Predicated Execution in Out-of-Order Processors. In *13th Int’l Symp. on High Performance Computer Architecture*, pages 75–84, 2007.
- [51] Amir Roth and Gurindar S. Sohi. Speculative Data-driven Multithreading. In *7th Int’l Symp. on High Performance Computer Architecture*, pages 37–48, 2001.

- [52] Yiannakis Sazeides, Andreas Moustakas, Kypros Constantinides, and Marios Kleanthous. The Significance of Affectors and Affectees Correlations for Branch Prediction. In *3rd Int'l Conf. on High Performance Embedded Architectures and Compilers*, pages 243–257, 2008.
- [53] André Seznec. Analysis of the O-GEometric History Length Branch Predictor. In *32nd Int'l Symp. on Computer Architecture*, pages 394–405, 2005.
- [54] André Seznec. The L-TAGE Branch Predictor. *The Journal of Instruction-Level Parallelism*, 2007.
- [55] André Seznec. A 64 Kbytes ISL-TAGE Branch Predictor. In *3rd Championship Branch Prediction*, 2011.
- [56] André Seznec. A New Case for the TAGE Branch Predictor. In *44th Int'l Symp. on Microarchitecture*, pages 117–127, 2011.
- [57] André Seznec, Stephen Felix, Venkata Krishnan, and Yiannakis Sazeides. Design Trade-offs for the Alpha EV8 Conditional Branch Predictor. In *29th Int'l Symp. on Computer Architecture*, pages 295–306, 2002.
- [58] André Seznec and Pierre Michaud. A Case for (Partially) TAGged GEometric History Length Branch Prediction. *The Journal of Instruction-Level Parallelism*, 2006.
- [59] Rami Sheikh, James Tuck, and Eric Rotenberg. Control-Flow Decoupling. In *45th Int'l Symp. on Microarchitecture*, pages 329–340, 2012.
- [60] Timothy Sherwood and Brad Calder. Loop Termination Prediction. In *3rd Int'l Symp. on High Performance Computing*, pages 73–87, 2000.
- [61] James E. Smith. A Study of Branch Prediction Strategies. In *8th Int'l Symp. on Computer Architecture*, pages 135–148, 1981.
- [62] James E. Smith. Decoupled Access/Execute Computer Architectures. In *9th Int'l Symp. on Computer Architecture*, pages 112–119, 1982.
- [63] Eric Sprangle, Robert S. Chappell, Mitch Alsup, and Yale N. Patt. The Agree Predictor: A Mechanism For Reducing Negative Branch History Interference. In *24th Annual Int'l Symp. on Computer Architecture*, pages 284–291, 1997.
- [64] Srikanth T. Srinivasan, Ravi Rajwar, Haitham Akkary, Amit Gandhi, and Mike Upton. Continual Flow Pipelines. In *11th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 107–119, 2004.
- [65] Standard Performance Evaluation Corporation. The SPEC CPU 2006 Benchmark Suite. In <http://www.spec.org>.
- [66] Renju Thomas, Manoj Franklin, Chris Wilkerson, and Jared Stark. Improving Branch Prediction by Dynamic Dataflow-based Identification of Correlated Branches from a Large Global History. In *30th Int'l Symp. on Computer Architecture*, pages 314–323, 2003.

- [67] Ross Albert Towle. *Control and Data Dependence for Program Transformations*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1976.
- [68] James Tuck, Wonsun Ahn, Luis Ceze, and Josep Torrellas. SoftSig: Software-Exposed Hardware Signatures for Code Analysis and Optimization. In *13th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 145–156, 2008.
- [69] Akhilesh Tyagi, Hon-Chi Ng, and Prasant Mohapatra. Dynamic Branch Decoupled Architecture. In *17th Int'l Conf. on Computer Design*, pages 442–450, 1999.
- [70] Neil Vachharajani, Ram Rangan, Easwaran Raman, Matthew J. Bridges, Guilherme Ottoni, and David I. August. Speculative Decoupled Software Pipelining. In *16th Int'l Conf. on Parallel Architecture and Compilation Techniques*, pages 49–59, 2007.
- [71] Bob Valentine. Introducing Sandy Bridge. Presented at Intel Developer Forum, San Francisco, 2010.
- [72] Sravanthi Kota Venkata, Ikkjin Ahn, Donghwan Jeon, Anshuman Gupta, Christopher Louie, Saturnino Garcia, Serge Belongie, and Michael Bedford Taylor. SD-VBS: The San Diego Vision Benchmark Suite. In *2009 Int'l Symp. on Workload Characterization*, pages 55–64, 2009.
- [73] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *22nd Int'l Symp. on Computer Architecture*, pages 24–36, 1995.
- [74] Tse-Yu Yeh and Yale N. Patt. Alternative Implementations of Two-Level Adaptive Branch Prediction. In *19th Int'l Symp. on Computer Architecture*, pages 124–134, 1992.
- [75] Craig Zilles and Gurindar Sohi. Execution-based Prediction using Speculative Slices. In *28th Int'l Symp. on Computer Architecture*, pages 2–13, 2001.

APPENDICES

Appendix A

Workloads with Predictable Control-Flow

We profiled *SPLASH-2* [73] applications using the methodology described in Section 2.1 of Chapter 2. All applications are run to completion using 1-thread and 4-threads. The profiles collected under both configurations are very much the same. i.e., branch predictability did not change when an application executed as a single- or multi-threaded application. Figure A.1 shows each application’s misprediction rate and mispredictions-per-1K-instructions (MPKI) for three branch predictors¹: BIMODAL [61], GSHARE [43], and state-of-art ISL-TAGE [55]. Interestingly, all *SPLASH-2* applications have predictable control-flow (especially when a state-of-art predictor is used).

Similarly, we profiled *San Diego Vision Benchmark Suite (SDVBS)* [72], *BYTEmark benchmark (nBench)* [42] and *Firefox* web browser. Figure A.2 shows the collected profiles. Again, all applications appear to have predictable control-flow.

¹All predictors are allocated a 64KB budget.

Workload	Application	BPKI	Misprediction Rate (%)			MPKI		
			BIMODAL	GSHARE	ISL-TAGE	BIMODAL	GSHARE	ISL-TAGE
SPLASH-2	barnes	51.09	6.51	4.68	1.70	3.33	2.39	0.87
	cholesky	62.10	4.63	2.87	1.19	2.88	1.78	0.74
	fft	50.29	8.34	1.40	0.02	4.20	0.71	0.01
	fmm	73.39	37.91	2.96	0.35	27.82	2.18	0.25
	lu_c	110.77	6.84	0.49	0.02	7.58	0.55	0.03
	lu_n	112.29	7.01	0.54	0.02	7.87	0.60	0.03
	ocean_c	59.89	2.03	1.13	0.03	1.22	0.68	0.02
	ocean_n	45.80	2.45	1.37	0.04	1.12	0.63	0.02
	radix	31.15	0.00	0.00	0.00	0.00	0.00	0.00
	raytrace	134.96	0.73	0.38	0.22	0.99	0.52	0.30
	water_n	90.97	20.51	2.11	1.39	18.65	1.92	1.26
	water_s	77.43	11.92	0.99	0.37	9.23	0.76	0.28

Figure A.1: SPLASH-2 profiles.

Workload	Application	BPKI	Misprediction Rate (%)			MPKI		
			BIMODAL	GSHARE	ISL-TAGE	BIMODAL	GSHARE	ISL-TAGE
SDVBS-1.3.2	disparity	80.53	0.49	0.43	0.38	0.40	0.34	0.31
	localization	223.82	0.75	0.03	0.03	1.67	0.08	0.07
	mser	90.61	16.86	5.13	0.91	15.27	4.65	0.82
	multi_ncut	199.98	0.00	0.00	0.00	0.00	0.00	0.00
	sif	147.85	2.38	1.61	0.01	3.53	2.37	0.01
	stitch	116.55	8.16	0.49	0.08	9.51	0.58	0.10
	svm	52.61	0.48	0.46	0.03	0.25	0.24	0.01
	texture_synthesis	143.92	4.16	2.26	0.19	5.98	3.25	0.27
	tracking	58.65	13.57	0.04	0.02	7.96	0.03	0.01
nBench-2.2.3	nbench	58.57	8.08	3.88	1.31	4.73	2.28	0.77
Web Browser	Firefox-17.0.1	6.15	4.71	3.12	2.15	0.29	0.19	0.13

Figure A.2: SDVBS, nBench and Firefox profiles.

Appendix B

Properties of the Control-Flow Constrained Applications

Figure B.1 shows three pieces of information for each control-flow constrained application.

1. Statistics gathered by running the application to completion¹: The misprediction rate, the frequency of branches (branches-per-kilo-instructions or BPKI) and the frequency of mispredictions (mispredictions-per-kilo-instructions or MPKI).
2. Statistics gathered over the regions simulated in our evaluation²: The frequency of cache misses for each cache level, namely, L1 cache MPKI, L2 cache MPKI and L3 cache MPKI.
3. Manual analysis information: The breakdown of MPKI according to the proposed/used solutions, namely: if-conversion, CFD, and VEC. The fraction of unhandled MPKI is also shown under inseparable and not-analyzed.

¹State-of-art ISL-TAGE [55] predictor is used as the default predictor.

²Refer to Tables 7.1 and 7.4 for more information.

Workload	Application	Run To Completion			Simulated Region			Analyzed Branches (Solution vs. Contribution)					
		Misprediction Rate (%)	BPKI	MPKI	L1 MPKI	L2 MPKI	L3 MPKI	If-Conversion	CFD (BQ)	CFD (TQ)	VEC	Inseparable	Not Analyzed
SPEC2006	astar (BigLakes #1)	7.67	131.86	10.11	118.96	34.30	13.41	3%	82%	12%	0%	0%	3%
	astar (BigLakes #2)				99.91	36.09	28.04						
	astar (Rivers #1)				82.38	12.53	1.15						
	astar (Rivers #2)	14.41	180.29	25.98	83.60	33.95	0.35	0%	97%	3%	0%	0%	0%
	bzip2 (chicken)	3.20	137.58	4.40	1.66	0.38	0.01	20%	3%	1%	1%	60%	15%
	bzip2 (input.source)	6.66	122.52	8.16	28.34	4.47	0.48	0%	5%	3%	25%	54%	13%
	gobmk	4.57	156.91	7.17	11.44	1.02	0.44	0%	0%	0%	0%	0%	100%
	gromacs	5.05	22.32	1.13	6.87	0.20	0.10	0%	40%	0%	0%	0%	60%
	mcf	4.97	182.11	9.06	189.92	76.41	41.08	0%	43%	0%	0%	35%	22%
	namd	3.25	36.14	1.17	10.23	0.66	0.70	0%	25%	0%	0%	0%	75%
	sjeng	2.75	187.42	5.15	2.73	0.29	0.25	22%	0%	0%	0%	35%	43%
	soplex (pds)	4.29	143.24	6.14	186.54	24.27	6.94	12%	20%	0%	0%	18%	50%
	soplex (ref)	1.97	114.32	2.25	69.44	10.39	10.59	0%	31%	0%	0%	59%	10%
	clustalw	3.50	121.31	4.25	0.01	0.00	0.01	100%	0%	0%	0%	0%	0%
	BioBench	fasta	10.82	156.22	16.90	0.95	0.04	0.04	80%	0%	0%	0%	0%
	hmmer	7.81	157.80	12.32	28.16	2.59	0.79	98%	0%	0%	0%	0%	2%
Nu-MineBench-3.0	ec1at	6.20	164.34	10.19	12.56	2.35	0.27	0%	100%	0%	0%	0%	0%
	gsm	2.77	75.96	2.10	0.02	0.01	0.01	70%	0%	0%	0%	7%	23%
cBench-1.1	jpeg-compr	7.25	112.70	8.17	1.54	0.12	0.09	25%	60%	0%	0%	0%	15%
	jpeg-decompr	4.55	53.01	2.41	0.10	0.02	0.03	55%	0%	0%	0%	26%	19%
	quick-sort	4.13	112.35	4.64	5.53	0.38	0.20	90%	0%	0%	0%	4%	6%
	tiff2bw	4.15	130.41	5.42	16.64	0.03	0.03	5%	40%	0%	0%	50%	5%
	tiff2rgba	4.35	127.08	5.53	30.05	0.04	0.05	5%	43%	0%	0%	50%	2%
	tiffdither	3.93	142.59	5.61	1.71	0.08	0.08	5%	42%	0%	0%	50%	3%
	tiffmedian	2.18	165.13	3.60	13.31	0.05	0.07	7%	60%	0%	0%	0%	33%

Figure B.1: Properties of the control-flow constrained applications.

Appendix C

Branch Queue Size Selection

Properly sizing the BQ is crucial for an efficient CFD implementation. In this appendix, we use *astar* (region #1) to demonstrate the process of selecting the BQ size.

Three components influence the BQ size:

1. *Window size*: Ideally, we want perfect BQ hit rate (i.e., 100%). We can guarantee this if the push and its corresponding pop are sufficiently separated from one another (i.e., they should be at least one-window instructions away from each other). For example, if each branch slice is 6 instructions long, and if we have a 168 instruction window. Then, the minimum BQ size required to provide the needed separation is: $168/6 = 28$ entries. Moreover, there can be at most 168 pending pushes in the window. Hence, there is no need for having a BQ larger than 168.
2. *Loop strip-mining overhead*: The BQ size impacts the loop strip-mining overhead as follows. The smaller the BQ, the more times we need to execute the strip-mining loop, the higher the instruction overhead, and the lower the CFD efficiency. This component favors having a larger BQ.
3. *Data locality*: In many cases, we find that data accessed by the first CFD loop are re-accessed by the second CFD loop. Reducing the distance between an iteration in the first

loop and its correspondence in the second loop will increase the probability of having cache hits. This component favors having a smaller BQ.

Figures C.1, C.2 and C.3 show the BQ miss rate, speedup, and instruction overhead as we increase the BQ size from 32 to 4096. Figure C.1 shows that we get a 100% BQ hit rate for BQ sizes of 64 and higher. Figure C.2 shows that performance peaks at BQ size of 128. Figure C.3 shows that we achieve minimum instruction overhead at BQ sizes: 128 and 256. Overall, a 128-entry BQ provides a sweet spot under which we deliver high performance and data locality, and low overheads.

Figure C.1: Impact of varying the BQ size on BQ miss rate.

Figure C.2: Impact of varying the BQ size on speedup.

Figure C.3: Impact of varying the BQ size on loop strip-mining overhead.

Appendix D

Design Space Exploration for Baseline

In an effort to find the best-performing baseline, we explored the design space of misprediction recovery policies. For simplicity and uniformity, all non-control-flow constrained applications (i.e., all applications except the ones listed in Figure B.1) are fast-forwarded for 10B instructions, while the control-flow constrained applications are fast-forwarded to the first occurrence of a function containing a hard-to-predict branch (as shown in Tables 7.1 and 7.4 in Chapter 7). After fast-forwarding, we warm-up for 10M and then simulate 100M instructions. Some of the applications run for less than 10B instructions; these are not fast-forwarded. Also, we exclude applications we could not compile with the Alpha gcc compiler.

The explored design space aspects are:

1. *Checkpoint management*: Three different aspects are considered: 1) Allocation: allocate a checkpoint for every branch vs. use confidence estimator [30]. 2) Reclamation: in-order vs. out-of-order. 3) Policy when no free checkpoints are left: stop dispatch vs. continue dispatch (hence no checkpoint allocated). The number of checkpoints is fixed at 8 for this study.
2. *Number of checkpoints*: 2 through 64, or none¹. The best configuration from #1 is used.

¹None corresponds to forward walking the ROB while copying the architectural map table (AMT) to the rename map table (RMT) in 8-cycles.

Figures D.1 and D.2 summarize the results of our exploration. Figure D.1 demonstrates that the best checkpoint management policy is: stop dispatch, OoO reclamation and with confidence estimator. Figure D.2 shows that the harmonic mean IPC, across all applications of all workloads, levels off at 8 checkpoints.

Figure D.1: Design space exploration: checkpoint management.

Moreover, we explored several misprediction recovery mechanisms:

1. Use checkpoints for recovery: Two configurations are shown, infinite checkpoints and finite checkpoints (8-checkpoints, our baseline).
2. Walk the ROB: Three approaches are evaluated. Always walk backward, always walk forward, or walk backward or forward depending on which scheme takes fewer cycles to complete.
3. Treat mispredictions as exceptions: Recover from the architectural state when the mis-

Figure D.2: Design space exploration: number of checkpoints.

Figure D.3: Design space exploration: misprediction recovery mechanism.

predicted branch reaches the head of the ROB.

The recovery mechanism exploration results are shown in Figure D.3. When walking the ROB is used, we observe that the average IPC is not degraded. This is expected since our baseline uses a fetch-to-rename pipeline depth of 6-cycles, which allows ample time for walking the ROB before the first instruction from the corrected path is renamed. Also, we observe a significant IPC degradation when mispredictions are treated as exceptions, which is expected as well.

Appendix E

Applying CFD to Predictable Branches

Figure E.1 shows a loop (*Original Loop*) transformed for CFD (*Decoupled Loops*). Referring to the original loop, the branch of interest (line 2) can be easy-/hard-to-predict depending on the initialization of array $a[]$ and the used branch predictor. We experiment with two initialization functions. We will refer to the two cases as microbenchmark #1 and microbenchmark #2, respectively. For both microbenchmarks, the number of retired instructions in the unmodified binaries (i.e., baseline) is 25M, and the CFD overhead is 1.12. Note that the control-dependent region contains two other branches (lines 5 and 6), each of which is strongly biased in one direction (T or NT), thus easy-to-predict by all predictors.

The arrays being accessed in the microbenchmarks are sized to fit in the L1 data cache. Thus, after the initialization phase (used for warm up in our simulations), the array accesses always hit in the cache.

For the purposes of this study, we will use three branch predictors: BIMODAL [61], G-SHARE [43], and state-of-art ISL-TAGE [55]. All predictors are allocated a 64KB budget.

1. Microbenchmark #1: The elements of array $a[]$ are initialized using the iteration number. Such initialization causes the condition $(a[i] \% 6)$ to have a repetitive pattern of: true (five times), then false (one time), and so on. This pattern cannot be perfectly predicted with BIMODAL, but can be perfectly predicted with a history-based predictor that utilizes

<i>Initialization Loop</i>	
1	for(int i=0; i<MAX; i++) {
2	//a[i] = i; // microbenchmark #1
3	//a[i] = rand(); // microbenchmark #2
4	b[i] = 0;
5	c[i] = 1;
6	d[i] = 1;
7	}
<i>Original Loop</i>	
1	for(int i=0; i<MAX; i++) {
2	if(a[i] % 6) { // hard-to-predict branch
3	b[i] = a[i] + 5;
4	b[i] *= 7;
5	if(c[i] > 0) c[i] += 3; // predictable branch (always NT)
6	if(d[i] == 0) d[i] = 5; // predictable branch (always T)
7	}
8	}
<i>Decoupled Loops</i>	
<i>First Loop</i>	
1	for(int i=0; i<MAX; i++) {
2	pred = (a[i] % 6); // the predicate is computed
3	Push_BQ(pred); // then pushed onto the BQ
4	}
<i>Second Loop</i>	
5	for(int i=0; i<MAX; i++) {
6	Branch_on_BQ { // pop the predicate
7	b[i] = a[i] + 5;
8	b[i] *= 7;
9	if(c[i] > 0) c[i] += 3; // predictable branch (always NT)
10	if(d[i] == 0) d[i] = 5; // predictable branch (always T)
11	}
12	}

Figure E.1: Microbenchmarks.

sufficient history. In our experiments, this repetitive pattern is successfully captured by ISL-TAGE but not G-SHARE.

2. Microbenchmark #2: The elements of array $a[i]$ are initialized using a random generator. Such initialization causes the condition $(a[i] \% 6)$ to have irregular, non-repetitive pattern, which cannot be perfectly captured by any of the used predictors.

Figure E.2 shows the MPKIs of the unmodified microbenchmarks. Our simulations show that: (1) The branches inside the control-dependent region (lines 5 and 6 of the original loop) are perfectly predicted by all predictors. (2) All mispredictions are caused by the branch of interest (line 2).

Figure E.3 shows the effective IPC of baseline and CFD for the three predictors. Observe that the CFD binaries deliver the same IPC independent of the underlying predictor.

Figure E.2: The MPKIs of the unmodified microbenchmarks using three branch predictors.

1. Microbenchmark #1: The branch of interest is perfectly predicted only by ISL-TAGE. In this case, CFD delivers performance equivalent to that of baseline with ISL-TAGE, and outperforms baseline with BIMODAL or G-SHARE.
2. Microbenchmark #2: The branch of interest is hard-to-predict by all predictors. In this case, CFD delivers improved performance compared to baseline with BIMODAL, G-SHARE, or ISL-TAGE.

Figure E.3: Effective IPC.

We conclude that: applying control-flow decoupling to branches with intricate patterns, that can be accurately predicted by large, complex, and specialized predictors, may enable substituting such predictors with smaller, simpler, and more generic predictors. That is, handling intricate branches with control-flow decoupling might make complex predictors redundant. Benefits of smaller and simpler predictors include higher clock frequency, lower energy consumption, and easier-to-pipeline branch prediction logic (for even higher frequency).