ABSTRACT

SRINIVASAN, VINESH. Slipstream Processors Revisited: Exploiting Branch Sets. (Under the direction of Eric Rotenberg and James Tuck.)

Delinquent branches (frequently mispredict) and loads (frequently miss) remain key IPC bottlenecks in some applications. One approach to reduce their effect is pre-execution via helper threads. Helper threads resolve delinquent branches and initiate delinquent loads before the main thread fetches corresponding instances of these instructions. Broadly, there are two classes of pre-execution: one class repeatedly forks small helper threads, each targeting an individual dynamic instance of a delinquent branch or load; the other class begins with two redundant threads in a leader-follower arrangement, and speculatively reduces the leading thread. The objective of this paper is to design a new pre-execution microarchitecture that meets four criteria: (i) retains the simpler coordination of a leader-follower microarchitecture as compared to per-dynamic-instance helper threads, (ii) is fully automated with just hardware, (iii) targets both branches and loads, (iv) and is effective. We review prior pre-execution proposals and show that none of them meet all four criteria. Using terminology from one of the first leader-follower processors (Slipstream), we propose Slipstream 2.0. The key innovation in the space of leader-follower architectures is to remove the *forward control-flow slices* of pre-executable delinquent branches and delinquent loads, from the leading thread. This new tactic is essential for the A-stream to perform: (1) Delinquent Branch Pre-execution (DBP): pre-execute hard-to-predict branches without getting bogged down by their misprediction penalties, (2) Delinquent Load Prefetch (DLP): effectively prefetch like post-slipstream closely related work, and (3) DBP+DLP: tolerate cache-missed loads that feed mispredicted branches, a first for kilo-instruction processor microarchitectures.

Slipstream 2.0 has the following advantages. It is the first pre-execution proposal that meets all four criteria; it includes a simple enable/disable mechanism based on profitability

making it a useful microarchitectural turbo-boost feature; it efficiently reduces the leading thread by a simple adaptation of conventional branching (“branch-to-reconvergent-point” instead of “branch-to-target”). Slipstream 2.0 is compared to the baseline single core (configured similarly to Intel Skylake) and the only other hardware-only leader-follower prior works in pre-execution: Slipstream (targets branches) and Dual Core Execution (DCE) (targets loads). For SPEC 2006/2017 SimPoints wherein Slipstream 2.0 is auto-enabled, it achieves geomean speedups of 67%, 60%, and 12%, over baseline, Slipstream, and DCE. For SimPoints with primarily delinquent branches, Slipstream 2.0 is 34%, 23%, and 21%, faster than baseline, Slipstream, and DCE. For SimPoints with primarily delinquent loads, Slipstream 2.0 is 84%, 73%, and 9%, faster than baseline, Slipstream, and DCE. It also demonstrates an average reduction of 43% in Energy Delay Product (EDP) and 4% in energy compared to baseline.

© Copyright 2019 by Vinesh Srinivasan

All Rights Reserved

Slipstream Processors Revisited: Exploiting Branch Sets

by
Vinesh Srinivasan

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

2019

APPROVED BY:

Huiyang Zhou

Rhett Davis

Eric Rotenberg
Co-chair of Advisory Committee

James Tuck
Co-chair of Advisory Committee

DEDICATION

To my family, Srinivasan Parthasarathy, Vijayavalli Venkatachary, Harish Srinivasan, and
Srividhya Jambunathan.

BIOGRAPHY

Vinesh was born in Chennai, India on 21st April 1992. He earned his Bachelors of Engineering in Electronics and Communication Engineering from Anna University, Chennai, India on May 2013. Vinesh was part of the high performance computer architecture group at Waran Research Foundation (WARFT, India) before joining NC State in Fall 2013. Soon after joining NC State, Vinesh started working as a Research Assistant under the supervision of Dr. Eric Rotenberg. His initial research project involved design and tapeout of a 3D processor designed for faster thread migration between dual cores. Vinesh received his Master of Science degree in Computer Engineering with the defense of his thesis on Aug 2015. Following his penchant towards high performance computer architecture, Vinesh decided to pursue his doctoral degree at NC State from Fall 2015. During the course of his graduate study, he successfully completed internships with AMD Research and product development teams. After the defense of his Ph.D., Vinesh will be joining the CPU performance team at Apple in Cupertino, California.

ACKNOWLEDGEMENTS

My journey through graduate school has been more like the Chinese bamboo. They say the tree remains a tiny green shoot for 5 years and grows into a full-fledged tree in a few months. For the first 5 years, the tree is forming a strong network of roots underground, to support the tree. In my case, it is the network of people who have been my support system without whom this dissertation will not be possible and I would like to acknowledge them.

First, I would like to thank my advisor Dr. Eric Rotenberg who has been a constant support from the moment I reached NC State. His course lectures laid the foundation for my research at NC State. He is hands down the best teacher a grad student can ask for. Thanks for all the effort he puts into preparing class materials, delivering the lecture, classroom interactions, and offline discussions. As an advisor, he has always been supportive and available when I need guidance. Also, thanks for the financial support and taking the burden away from your students.

Secondly, I would like to thank Dr. James Tuck, my co-advisor. When navigating through research can sometimes be daunting and uncertain he has always been a source of confidence. I want to thank him for all the research and tangent technical discussions which made me a better computer architecture researcher. I also want to thank my doctoral committee members Dr. Rhett Davis, Dr. Huiyang Zhou, and Dr. Ashok Gopalarathnam for your time and valuable comments during the presentations that helped me improve the work. I would also like to thank all ECE faculty who I have had classes with, I learned a lot from them. Also thanks to all the ECE graduate office staff for helping me out with all the administrative formalities.

My family has always provided their unconditional support through the course of my graduate study. I would like to thank my parents, Srinivasan Parthasarathy and Vijayavalli Venkatachary for all the sacrifices they make for me and giving me the freedom to make my own decisions. My brother Harish Srinivasan for always being there for me. My wife

Srividhya Jambunathan for patiently waiting and supporting me through some difficult times during my Ph.D. I am forever grateful to them.

I was fortunate to meet some passionate and hard-working fellow graduate students who made a positive impact on me. I would like to thank Rangeen Basu Roy Chowdhury for being a mentor. His mentorship made me a better researcher and computer engineer. Thanks for all the phone discussions and hard work that resulted in successful paper submissions and project milestones. I would also like to thank Elliott Forbes, Amro Awad, Abhinav Agrawal, Randy Hari Widialaksono, Bagus Wibowo, and Sungkwan Ku for all the technical discussions and professional advice during my Ph.D. Also, big thanks to Anil Krishna, Anil Kannepalli, and Muawya Al-Otoom for your timely advice and guidance. Thanks to Chanchal Kumar for your help with this work.

I owe most of my personal and professional growth to my undergraduate advisor Prof. Venkateswaran. He triggered the curiosity and drive needed for research and I owe a huge acknowledgment to him for encouraging me towards the Ph.D. path. I would like to thank my WARFT senior Nachiappan Chidambaram for his guidance, my WARFT buddies Ram Srivatsa and Prashanth Thina for their company. My undergrad friends Vignesh and Varshad for being by my side through thick and thin. My life in Raleigh and in turn my Ph.D. wouldn't have been enjoyable without the friends and the awesome people that I had the opportunity to spend time with. Thanks to Vaishnavi Kumar for being there for me as a friend during some tough times, Arun Vishnu, Anusha, Radha, Sivashankari, Moses, and Hari for some of the most memorable times of my life in Raleigh. I hope our friendship continues to the next chapter of my life as well.

This dissertation was supported in part by NSF grant no. CCF-1823517, and grants from Intel. Any opinions, findings and conclusions or recommendations expressed herein are those of the author and do not necessarily reflect the views of the National Science Foundation or Intel.

TABLE OF CONTENTS

LIST OF TABLES	viii
LIST OF FIGURES	ix
LIST OF LISTINGS	xi
Chapter 1 Introduction	1
1.1 Motivation	1
1.2 Pre-execution, Our Objectives, and Past Works Measured by These Objectives	3
1.3 Slipstream 2.0 and Thesis Contributions	7
1.4 Dissertation Outline	11
Chapter 2 Novelty and Related Work	13
2.1 DCE, CFP, and Runahead	14
2.2 Decoupled Look-Ahead Architecture	16
2.3 Pre-Execution, Helper Threads	17
Chapter 3 Slipstream Processor 2.0	19
3.1 Original Slipstream Microarchitecture	19
3.2 Slipstream 2.0 Microarchitecture	21
3.2.1 IR-detector	23
3.2.2 IR-predictor	28
3.2.3 Delay Buffer	29
3.2.4 Storage cost	30
Chapter 4 Case Studies	31
4.1 Delinquent Branch Pre-execution	31
4.1.1 Case Study: ASTAR	32
4.1.2 Case Study: hmmer	33
4.2 Delinquent Load Prefetch	34
4.2.1 Case Study: libquantum	35
4.2.2 Case Study: mcf	36
Chapter 5 Reconvergence Predictor	38
5.1 Reconvergence Predictor Example	40
Chapter 6 Branch Set Analysis Example	44
Chapter 7 Evaluation	51
7.1 Benchmark Analysis	51
7.2 Simulator Configuration	51
Chapter 8 Results	53
8.1 Delinquent Branch Pre-execution (DBP)	53

8.1.1	Speedup	53
8.1.2	Number of Delinquent and Pre-executable Branches	55
8.1.3	Branch Set Analysis	56
8.2	Delinquent Load Prefetching (DLP)	57
8.2.1	Speedup	57
8.2.2	Number of Delinquent Loads	58
8.2.3	Branch Set Analysis	59
8.3	Delinquent Load Prefetching (DLP) + Delinquent Branch Pre-execution (DBP)	61
8.3.1	DBP+DLP Proactive	61
8.3.2	DBP+DLP Reactive	61
8.3.3	Delinquent Branches and Loads in DBP+DLP	63
8.4	Comparisons with Slipstream 1.0 and DCE	66
8.5	Microarchitectural Turbo Boost	67
8.6	IR-mispredictions	69
8.7	Delay Buffer Occupancy	70
Chapter 9	Energy	72
Chapter 10	Conclusion	74
BIBLIOGRAPHY		75

LIST OF TABLES

Table 1.1	Past pre-execution proposals characterization	5
Table 2.1	Comparison with prior art	14
Table 3.1	Branch Load Classifier - (49 bits x 128 entries) + (23 bits x 8 entries) .	26
Table 3.2	Branch Set Buffer - 1025 bits x 1 entry = 128B	28
Table 3.3	Data Dependence Tracker - 64 Bits Total	28
Table 3.4	IR Predictor Entry - 79 bits x 128 entries	29
Table 5.1	Formats of the various tables in the Reconvergence Predictor.	39
Table 7.1	Core configuration with parameters modelled after Intel Skylake Desktop Processor, for in-common major superscalar parameters. .	52

LIST OF FIGURES

Figure 1.1	SPEC 2006 (no _s extension) and 2017 (_s extension) benchmarks analyzed.	2
Figure 1.2	ILP potential in target benchmarks.	3
Figure 1.3	Example of a delinquent branch and its forward control-flow slice.	10
Figure 3.1	Slipstream Processor. Shaded components and dotted lines represent the newly added structures and their connectivity	20
Figure 3.2	Slipstream Processor 2.0 Microarchitecture. Shaded components and dotted lines represent the newly added structures and their connectivity	22
Figure 3.3	IR-detector.	23
Figure 5.1	On-going Reconvergence Predictor example (snapshot 1).	41
Figure 5.2	On-going Reconvergence Predictor example (snapshot 2).	42
Figure 5.3	On-going Reconvergence Predictor example (snapshot 3, final).	43
Figure 6.1	Branch Set Analysis example (snapshot 1).	45
Figure 6.2	Branch Set Analysis example (snapshot 2).	46
Figure 6.3	Branch Set Analysis example (snapshot 3).	47
Figure 6.4	Branch Set Analysis example (snapshot 4).	48
Figure 6.5	Branch Set Analysis example (snapshot 5).	49
Figure 6.6	Branch Set Analysis example (snapshot 6, final).	50
Figure 8.1	Speedups of DBP over baseline, and baseline with perfect branch prediction.	54
Figure 8.2	Delinquent branches and CIDI branches observed in DBP.	55
Figure 8.3	Delinquent branches' branch set sizes observed in DBP.	57
Figure 8.4	Speedups of DLP over baseline, and baseline with perfect data caches.	58
Figure 8.5	Delinquent Loads observed in DLP.	59
Figure 8.6	Delinquent loads' branch set sizes observed in DLP.	60
Figure 8.7	Speedups of DBP+DLP proactive and reactive over baseline, and baseline with perfect branch prediction and data caches.	62
Figure 8.8	Delinquent Branches and Loads observed in DBP+DLP.	63
Figure 8.9	Delinquent branches' branch set sizes observed in DBP+DLP reactive.	64
Figure 8.10	Delinquent loads' branch set sizes observed in DBP+DLP reactive.	65
Figure 8.11	Comparisons among Slipstream 1.0, DCE, and Slipstream 2.0 (DBP, DLP, and DBP+DLP).	66
Figure 8.12	Speedup of DBP+DLP over baseline, over Slipstream 1.0, and over DCE. Data is categorized into control-intensive, memory-intensive, and all benchmaks.	67
Figure 8.13	Slipstream 2.0 as microarchitectural turbo boost.	69
Figure 8.14	IR-Mispredictions per 1K instructions (MPKI) in Slipstream 2.0.	70
Figure 8.15	Delay Buffer Occupancy in Slipstream 2.0.	71

Figure 9.1 Energy and EDP of Slipstream 2.0 (dual cores) normalized to base-
line (single core). 73

LIST OF LISTINGS

Listing 4.1	astar makebound2 function	32
Listing 4.2	hammer P7Viterbi function	34
Listing 4.3	libqunatum quantum_toffoli function	35
Listing 4.4	mcf primal_bea_mpp function	36
Listing 8.1	Heuristics to turn on/off A-stream	67

CHAPTER

1

INTRODUCTION

1.1 Motivation

Applications continue to demand more compute performance. The slowing of frequency scaling necessitates looking at all forms of parallelism for performance. With Thread Level Parallelism (TLP) and Data Level Parallelism (DLP) having been successfully exploited for performance of late, Instruction Level Parallelism (ILP) has lagged and needs another look in the modern context. State-of-the-art branch predictors are more accurate and prefetchers can capture more complex address patterns than older counterparts. Despite this, branch mispredictions and cache misses remain the major limiters for single-thread performance. Static loads that cause the most cache misses are called delinquent loads. Data-dependent static branches that cause the most mispredictions are called delinquent branches. Often,

there is a cache-missed load feeding a mispredicted branch that neutralizes the latency hiding ability of large window processors, as all the instructions fetched in the shadow of the miss are squashed.

Figure 1.1 shows instructions-per-cycle (IPC) of the top-weighted SimPoints for the SPEC 2006 and SPEC 2017 benchmarks compiled to the RISC-V ISA. The baseline core uses a 5.5KB VLDP prefetcher [She15] and a 64KB LTAGE branch predictor [Sez16]. The maximum possible IPC for this 4-wide fetch/retire core is 4 IPC.

Figure 1.1 SPEC 2006 (no _s extension) and 2017 (_s extension) benchmarks analyzed.

Figure 1.2 shows IPCs for (1) the baseline core and (2) the same baseline core with perfect branch prediction and perfect L1 data cache (loads/stores always hit). Results are shown for some of the lowest-IPC benchmarks (below the 1.5 IPC line in Figure 1.1, more than 60% below the peak IPC of 4). All of them show more than 2x upper-bound speedup potential.

Figure 1.2 ILP potential in target benchmarks.

1.2 Pre-execution, Our Objectives, and Past Works Measured by These Objectives

One approach to target delinquent branches and loads is to exploit some form of pre-execution via helper threads. Helper threads resolve delinquent branches and initiate delinquent loads before the main thread fetches corresponding instances of these instructions.

Broadly, there are two classes of pre-execution. One class repeatedly forks small helper threads, each targeting an individual dynamic instance of a delinquent branch or load. A helper thread is the backward slice of instructions leading to the branch/load. The other class begins with two redundant threads in a leader-follower arrangement. The leading thread is speculatively reduced by pruning instructions. Pruning is such that the leading thread still maintains accurate overall control-flow.

The leader-follower class is attractive because coordinating the leader and follower is simple [GH08]. The leader is always active (although at a higher level it can be enabled only for profitable phases, as developed in this paper) and there is a one-to-one control-flow

correspondence between the leader and follower. Thus, it avoids tricky issues of the first class: it does not need careful timing of forking of per-dynamic-instance helper threads and careful alignment of each pre-executed branch outcome to the corresponding branch in the main thread.

The objective of our work is to propose a pre-execution microarchitecture that:

1. retains the simple coordination of a leader-follower microarchitecture,
2. is fully automated with just hardware,
3. targets both branches and loads,
4. and is effective.

To motivate our approach, we characterize past pre-execution proposals in Table 1.1 in terms of our four criteria, above.

Slice processor [Mos01], speculative precomputation [Col01], and continuous runahead [Has16]. These meet criteria 2 and 4: they are fully automated with just hardware and they are effective at what they target. They do not meet criteria 1 and 3, however: they are not leader-follower and they only target loads.

Speculative data-driven multithreading (DDMT) [RS01], execution-based prediction using speculative slices [ZS00], and simultaneous subordinate microthreading (SSMT) [Cha99; Cha02]. These approaches do not meet criteria 1 and 2: they are not leader-follower and they are not fully automated in hardware. For DDMT and speculative slices, backward slices of branches and loads are manually identified, and their trigger (fork) instructions are inserted manually in the main thread. SSMT is fully automated via a compiler: for some industry R&D CPU teams, compiler support for a microarchitecture addition is undesirable as it introduces a dependence between the two that is both difficult to justify and challenging to deploy. These approaches meet criteria 3 and 4: they target both branches and loads, and in principle there are no fundamental limitations to their efficacy as compared to competing approaches.

Table 1.1 Past pre-execution proposals characterization

Prior work	Criterion 1: leader-follower	Criterion 2: fully automated in hardware	Criterion 3: targets both branches and loads	Criterion 4: effective
Slice processor [Mos01], speculative precomputation [Col01], and continuous runahead [Has16]	no	yes	no (loads only)	yes
Speculative data-driven multithreading (DDMT) [RS01], execution-based prediction using speculative slices [ZS00] simultaneous subordinate microthreading (SSMT) [Cha99; Cha02]	no	no (manual or compiler)	yes	yes
Slipstream processor [Sun00; Pur00; Red06]	yes	yes	no (branches only)	no (limited branch pre-execution)
Dual core execution (DCE) [Zho05]	yes	yes	no (loads only)	yes, with caveat (load -> misp. br.)
Decoupled look ahead (DLA) [GH08; PH14; KH18b]	yes	no (tool)	yes	branches: no (limited branch pre-execution) loads: yes, with caveat (load -> misp. br.)

Slipstream processor [Sun00; Pur00; Red06]. Among the earliest leader-follower architectures, slipstream meets criterion 1. It also meets criterion 2: it is fully automated with hardware. To understand slipstream in the context of criteria 3 and 4, we explain more about how slipstream works. A slipstream processor runs two redundant copies of a program in a leader-follower arrangement on dual superscalar cores (or dual threads in a simultaneous multithreading core) to improve single-thread performance and fault tolerance. The leader (advanced-stream or A-stream) is speculatively reduced by removing confidently predicted branches and their backward slices, which are replaced by the confident predictions. The follower (redundant-stream or R-stream) receives a complete branch history from the A-stream: (i) original predictions for removed confident branches

and (ii) pre-executed outcomes for not-removed unconfident branches. We can conclude that, in general, slipstream does not meet criterion 4: in phases with mostly unpredictable branches, where branch pre-execution matters most, slipstream fails to prune enough instructions from the leading thread to be effective. Slipstream’s primary pruning criterion is to replace highly confident branches and their backward slices with confident predictions. This criterion is only useful when there is a balanced interleaving of confident and unconfident branches. This conclusion is confirmed in the results section. Slipstream also does not meet criterion 3: it targets branches but not loads. Slipstream’s backward slice removal does not stop short at delinquent loads transitively feeding the confident branches, losing out on the opportunity to convert these now-dead loads into non-binding prefetch instructions and exploit high memory level parallelism.

Dual core execution (DCE) [Zho05] . DCE uses dual redundant threads like slipstream and is fully automated in hardware, meeting criteria 1 and 2. It does not prune any instructions in the leading thread, *per se*. Instead, cache-missed loads that would otherwise block retirement in the leading thread for many cycles, and the loads’ forward slices, are pseudo-retired (load and its dependent instructions are unstalled and their invalid results discarded). That is, long-latency loads are dynamically converted to non-binding prefetches in the leading thread. DCE does not meet objective 3, however: it only targets loads. Thus, for phases heavy on delinquent branches and light on delinquent loads, DCE offers no speedup. DCE meets objective 4, but with an important caveat: it is highly effective in tolerating long-latency loads, as long as they do not have any mispredicted branches in their forward slices. For a pseudo-retired load, its dependent branches must also be pseudo-retired (owing to the load not forwarding a value), hence, resolution of these miss-dependent branches is deferred to the trailing thread. If the trailing thread detects a mispredicted branch, the leading thread is squashed and restarted from the trailing thread: the latency of the load is not hidden because this latency is transferred to the branch’s misprediction penalty. In terms of performance, it’s similar to the load simply blocking retirement for the duration

of the miss (actually it resembles classic runahead [Mut03] since at least other loads are initiated in the shadow of the squash).

Decoupled look ahead (DLA) [GH08; PH14; KH18b]. DLA generalizes dual redundant threads by combining concepts from slipstream and dual core execution. Confident branches, as determined by offline profiling, and their backward slices are replaced by unconditional branches (akin to slipstream). Delinquent loads, as determined by offline profiling, are reintroduced as non-binding prefetches if they were initially removed by the confident branch pruning. As a combination of slipstream and DCE, DLA meets criteria 1 and 3: it exemplifies leader-follower and it targets both branches and loads. DLA does not meet objective 2: it uses an offline tool to prune instructions from the binary to generate the leading thread's skeleton. Objective 4 is mixed: (i) like DCE, it is highly effective in targeting delinquent loads that feed predictable branches; (ii) like DCE, it cannot hide the latency of a cache-missed load that feeds a mispredicted branch; (iii) like slipstream, branch pre-execution breaks down when it matters most – DLA cannot achieve sufficient pruning for phases with mostly unpredictable branches.

1.3 Slipstream 2.0 and Thesis Contributions

Inspired by slipstream, our approach begins with dual redundant threads (A-stream, R-stream) in a leader-follower arrangement for simple coordination. Rather than remove backward slices of confident branches in the A-stream, the key idea is to remove *forward control-flow slices* of hard-to-predict pre-executable branches and delinquent loads in the A-stream while still ensuring correct control-flow overall. The forward control-flow slice of a delinquent branch includes its control-dependent region, as well as control-independent data-dependent branches with respect to the branch and their control-dependent regions. (That of a delinquent load is similar except it does not have its own control-dependent region.) A branch is *pre-executable* if it does not depend on itself: its forward control-flow

slice can be removed and the next dynamic instance of the branch will still execute correctly.

We propose a key concept called *branch sets*. A branch set is a list of control-independent data-dependent (CIDD) branches, with respect to a pre-executable branch or a load. Branch sets are important for two reasons. First, a branch is pre-executable if it is not in its own branch set. Second, the branch set describes the forward control-flow slice to be removed.

We propose a new slipstream processor that automatically identifies branch sets. The responsible hardware unit:

1. identifies delinquent branches and loads,
2. identifies branches' probable reconvergent points [Col04], to delineate branches' control-dependent (CD) regions, and
3. identifies branches'/loads' probable CIDD branches using simple forward poisoning of CD regions and CIDD instructions.

Hard-to-predict branches that are not in their own branch sets are identified for *Delinquent Branch Pre-execution (DBP)*. For DBP, the A-stream's fetch unit:

- i. fetches and retains the delinquent branch for pre-execution,
- ii. skips over the delinquent branch's CD region, and,
- iii. for each branch in the delinquent branch's branch set, fetches then discards the branch and skips over its CD region.

An example of a delinquent branch ① and its forward control-flow slice is shown in Figure 1.3. Its CD region ② has a potential write to r4 ③. Because branch ④ depends on r4, it is CIDD with respect to the delinquent branch. The CD regions of both branches ① and ④ have potential writes to r5, ⑤ and ⑥. Because branch ⑦ depends on r5, it is also CIDD (directly via branch-①/write-⑤ and transitively via branch-④/write-⑥). The forward control-flow slice of the delinquent branch is comprised of its CD region ②, its branch set ④

and ⑦, and the CD regions ⑧ and ⑨ of its branch set. In order for the A-stream to effectively pre-execute the delinquent branch, it needs to remove its entire forward control-flow slice. This leaves only the black boxes, labeled A, B, C, and D, including the delinquent branch and excluding branches in its branch set. Doing this means that the delinquent branch need not be predicted and any potential misprediction penalty is avoided. The delinquent branch and its entire forward control-flow slice is replaced with a predicate computation. The R-stream receives this predicate computation as a highly accurate branch prediction. The R-stream fetches and executes only the correct CD path of the delinquent branch, penalty-free (except for any local mispredictions of branches nested within the skipped CD region), and locally predicts and resolves its branch set and branch set's CD regions. *Summing up, the R-stream receives a pre-executed outcome for the delinquent branch and the A-stream is insulated from the R-stream's local resolution of deferred dependent gaps.*

Loads that frequently miss in the L1 and L2 caches are identified for *Delinquent Load Prefetching (DLP)*. For DLP, the A-stream's fetch unit:

- i. fetches the delinquent load and converts it to a non-binding prefetch instruction, and
- ii. for each branch in the delinquent load's branch set, fetches then discards the branch and skips over its CD region.

The R-stream receives all A-stream-executed branch outcomes as accurate predictions, but now any missing control-flow is the responsibility of the R-stream to flesh-out using its branch predictor plus execution. Missing control-flow includes discarded branches from the branch sets as well as branches nested within skipped CD regions. Note that, although there may be localized gaps between branches and their reconvergent points (which delineate CD regions), there is still a one-to-one correspondence in global control-flow between the A-stream and R-stream.

Slipstream 2.0 meets all four criteria of Section 1.2, unlike past pre-execution work: (1)

Figure 1.3 Example of a delinquent branch and its forward control-flow slice.

simple coordination of leader-follower style pre-execution, (2) fully automated in hardware, (3) targets both branches and loads, and (4) effective. It effectively pre-executes branches that are pre-executable (not self-dependent), even in regions of mostly unpredictable branches (addressing efficacy weaknesses of original slipstream and DLA). It tolerates long-latency loads that feed mispredicted branches, by insulating the A-stream from the now-localized misprediction recovery in the R-stream (addressing the load efficacy caveat of DCE and DLA).

Another nice aspect of Slipstream 2.0 is that instruction removal in the A-stream “looks like” traditional branching. Rather than prune individual arbitrary instructions, as slip-

stream and DLA does, Slipstream 2.0 skips entire CD regions, and *only* CD regions, by converting their branches to “branch-to-reconvergent-point” instead of “branch-to-taken-target”.

Finally, another contribution of this work is a hardware mechanism for enabling/disabling Slipstream 2.0 for phases during which it is profitable/unprofitable, respectively. Thus, Slipstream 2.0 can be used as a microarchitectural turbo-boost performance mode.

Results: First, the turbo-boost feature successfully enables/disables Slipstream 2.0 for benchmark SimPoints according to need. We find that, for most SimPoints, turbo-boost is either enabled (8 out of 25 SimPoints) or disabled (12 out of 25 SimPoints) for almost the entire SimPoint. Second, for the SimPoints where Slipstream 2.0 is enabled, geomean speedups of 67%, 60%, and 12%, are observed over the baseline, slipstream, and DCE, respectively. For benchmarks with primarily delinquent branches, Slipstream 2.0 is 34%, 23%, and 21%, faster than baseline, slipstream, and DCE, respectively. For benchmarks with primarily delinquent loads, Slipstream 2.0 is 84%, 73%, and 9%, faster than baseline, slipstream, and DCE, respectively. Third, Slipstream 2.0 utilizes 4% lesser energy and 43% lesser EDP compared to baseline. Note that these results are for two cores for Slipstream 2.0 versus only a single core for baseline. These results are due to shorter execution time decreasing energy despite redundant core activity, redundant core activity doesn't extend much beyond the two cores' L1 caches, and turbo-boost ensures energy is not wasted in unprofitable phases.

1.4 Dissertation Outline

The next chapter discusses closely related work. This discussion provides context for understanding how branch sets and the new A-stream criteria for DBP and DLP are novel and address key scenarios that defy past microarchitectures. *In particular, the next section emphasizes the essential feature whereby the A-stream skips branches and their CD regions,*

that transitively depend on delinquent branches and loads, and defers them to the R-stream.

Chapter 3 discusses the design of Slipstream Processors 2.0. Chapter 4 provides case studies, taken from SPEC benchmarks, to gain insight into the benefits of Slipstream Processors 2.0. Chapter 5 explains the reconvergence predictor with an example. Chapter 6 describes the branch set analysis with an example and how it is crucial in the implementation of DBP and DLP. Chapter 7 describes the simulation infrastructure. Chapter 8 discusses results and compares the proposed design with prior proposals. Chapter 9 discusses energy impact of Slipstream 2.0 and the dissertation is concluded in chapter 10.

CHAPTER

2

NOVELTY AND RELATED WORK

This work leverages classical slipstream’s underlying execution model and expands its utility for hard-to-predict and/or miss-bound program phases. The original slipstream processor is discussed in detail in Section 3.1.

The table 2.1 compares closely related prior work with Slipstream processors 2.0.

Table 2.1 Comparison with prior art

design	backward slice extraction				explicit timing of fork- ing		explicit branch align- ment		branch misp. tolerance	load miss tolerance
	manual	compiler	hardware	not needed	trigger instruc- tion	not needed	integration hard- ware	not needed		
slice processor [Mos01], speculative precomputation [Col01], continuous runahead [Has16]			✓		✓			✓		✓
simultaneous subordinate microthreading (SSMT) [Cha99; Cha02]		✓			✓		✓		✓	✓
speculative data-driven multithreading [RS01], execution-based prediction using speculative slices [ZS00]	✓				✓		✓		✓	✓
dual-core execution (DCE) [Zho05], continual flow pipeline (CFP) [Sri04]				✓ ¹		✓		✓		✓except load->misp
decoupled look-ahead (DLA) [GH08; PH14; KH18b]	✓(EDA style tool)					✓		✓	✓lead- ing thread is bot- tleneck in heavy misp. regions	✓except load->misp
slipstream processor 1.0 [Sun00; Pur00; Red06]			✓			✓		✓	✓A-stream is bottleneck in heavy misp. re- gions	
slipstream processor 2.0				✓ ¹		✓		✓	✓	✓

¹ simpler forward poisoning instead

2.1 DCE, CFP, and Runahead

Dual-Core Execution (DCE) [Zho05], like classical slipstream, runs two redundant threads on dual cores to improve single-thread performance and fault tolerance. Unlike classical slipstream, DCE intentionally views the dual cores and the branch queue between them as a kilo-instruction window for memory latency tolerance. It does this by pseudo-retiring a cache-missed load in the leading thread if it would otherwise cause retirement to stall for a long time. The load's dependent instructions are marked as poisoned and also pseudo-retired. This effectively converts long-latency loads into prefetches. The leading thread passes all branch outcomes, some of which are unverified as they correspond to poisoned branches, to the trailing thread. A branch misprediction detected in the trailing thread implies a load-miss-dependent mispredicted branch. Latency tolerance is lost in this scenario as the leading thread is considered off-track, squashed, and restarted from the trailing

thread.

The Continual Flow Pipeline (CFP) [Sri04] is similar to DCE except it uses a single core. For CFP on top of a ROB-based superscalar core, when an L2-missed load reaches the head of the ROB, a register checkpoint is taken and the load and its data-dependent slice (along a specific speculative path) are drained from the ROB along with the slice’s available live-in register values. When the load’s data becomes available, the deferred slice is reinjected for “post-execution” and its live-out values are merged into the now-far-ahead core’s state. If a late branch misprediction is detected in the deferred slice, the core must be rolled-back to the checkpoint. Thus, as with DCE, latency tolerance is lost in the case of a load-miss-dependent mispredicted branch. Indeed, this scenario was emphasized by the CFP authors as the last remaining bottleneck of this kilo-instruction microarchitecture.

In contrast to DCE and CFP, our proposed slipstream processor’s DLP technique provides latency tolerance even for cache-missed loads feeding mispredicted branches, owing to the intentional skipping of the load-miss-dependent branch’s control-dependent (CD) region. The A-stream initiates a prefetch for the load and defers the binding load instruction and its forward slice, including any dependent branches and their CD regions, to the R-stream. The R-stream locally fleshes-out the omitted load-miss-dependent control-flow without rolling back the A-stream.

Runahead Execution [Mut03] is similar to CFP in that it takes a register checkpoint when the missed load reaches the head of the ROB and drains the load and its data-dependent slice. It discards the slice, however. When the load’s data becomes available, the now-far-ahead core is unconditionally rolled-back to the checkpoint. The advantage of running ahead with the intention of rolling back is expanding the small ROB’s scope for prefetching other misses during the first load miss, getting multiple misses for the latency of one miss. This is a simpler microarchitecture than CFP but it cannot tolerate the first miss or an isolated miss. In any case, the rollback is unconditional whether or not there are mispredicted branches in the load’s slice.

Continuous Runahead Processor (CRE) [Has16] improves on Runahead by executing the speculative thread on a runahead engine for continuous execution. Like the other prefetching schemes, CRE won't benefit applications with higher branch mispredicts and for applications where loads feed these mispredicts.

Another advantage of the proposed slipstream processor's DLP, as compared to DCE, CFP, and Runahead, is that it does not require the complexity of propagating poison bits among in-flight instructions deep within the core. Dependence information is used in the IR-detector's data dependence tracker to train branch sets. Later, branch sets are used within the fetch unit to fetch and discard branches and skip to their reconvergent points.

2.2 Decoupled Look-Ahead Architecture

Garg and Huang [GH08] proposed the *performance-correctness explicitly-decoupled architecture*, an all-encompassing view of generating a much-reduced program skeleton, run on the performance core, to drive excellent branch prediction and prefetching for a full copy of the program, run on the correctness core. The skeleton is generated by a profiling compiler based on interesting criteria:

1. Biased branches and their backward slices are replaced with special unconditional branches (analogous to classical slipstream's replacing branches and their backward slices with confident predictions). This maintains the simple one-to-one branch alignment between the performance and correctness cores (FIFO branch outcome queue).
2. Store instructions and their backward slices are removed, if their dependent loads are sufficiently distant in the dynamic instruction stream, such that the correctness core is likely to have performed the same store that was removed from the skeleton before the now-speculative load is fetched in the skeleton. Store removal is necessary for significant reduction of the skeleton, in general, and is analagous to omitting stores

from pre-execution slices / helper threads (which are discussed later in this section).

3. If delinquent loads (loads that miss a lot) and their slices were pruned by the criteria, above, they are reinserted and converted to prefetch instructions. This is a software implementation of DCE-style prefetching.

The slipstream processor proposed herein adds novel skeleton-program criteria based on branch sets. The removal of forward slices of hard-to-predict pre-executable branches, *including their CD regions and the CD regions of CIDD branches*, is essential for pre-executing these branches without bogging down the A-stream with their misprediction penalties. Moreover, as was discussed above for DCE and CFP, *our consideration of CIDD branches for delinquent loads* allows for first-of-a-kind latency tolerance for delinquent loads feeding mispredicted branches.

In addition, the slipstream processor proposed herein identifies and exploits branch sets dynamically in hardware.

Parihar and Huang [PH14] applied genetic algorithms to further reduce the skeleton. This trial-and-error, profit-centric approach, can identify weak dependencies for further pruning opportunity. R3-DLA [KH18b] improves the efficiency of Decoupled Look ahead Architecture by further optimizing the skeleton’s execution resources but still the skeleton generation is done offline with the help of a profiling tool.

2.3 Pre-Execution, Helper Threads

There is much prior art on surgical pre-execution, helper threads, and scout threads, such as the work by Zilles and Sohi [ZS00; ZS01], Roth and Sohi [RS01], Chappell et al. [Cha99; Cha02], Collins et al. [Col01], and Moshovos, Pnevmatikatos, and Amirali [Mos01].

These microarchitectures must provide mechanisms for one or more of the following: (1) statically or dynamically constructing the backward slices of delinquent loads, delinquent branches, or both; (2) timing the forking of slices sufficiently ahead of when their outcomes

are needed, including in some proposals supplying value predictions for slice live-ins; (3) surgically aligning partial branch outcomes from the scout thread to counterparts in the main thread.

The slipstream processor proposed herein (and its progenitor, classical slipstream) leverages the simplicity of starting from a continuous redundant thread and pruning instructions. It does not require slicing hardware for extracting and caching the delinquent branch/load backward slices, does not require explicit timing of forking (owing to a continuous A-stream and reforking at infrequent IR-mispredictions in the R-stream), does not require slice live-in value predictions, and so forth.

CHAPTER

3

SLIPSTREAM PROCESSOR 2.0

3.1 Original Slipstream Microarchitecture

The original slipstream processor [Sun00; Pur00; Red06] executes the A-stream and R-stream on either dual cores or dual threads of a simultaneous multi-threading core. The variant depicted in Figure 3.1 uses dual cores. The components in the dotted lines were only part of the original slipstream. The instruction-removal method described here is the simplified iterative approach [KR04]. The instruction-removal detector (IR-detector) monitors the retired instruction stream of the R-stream and signals the instruction-removal predictor (IR-predictor), which is PC indexed table in the fetch stage, to increment or reset confidence counters for each dynamic instruction. Counters are incremented for correctly-predicted branches. Counters are also incremented for register-producing instructions that are deter-

Figure 3.1 Slipstream Processor. Shaded components and dotted lines represent the newly added structures and their connectivity

mined to be dynamically dead within a certain scope, *i.e.*, the producer's logical register is written by another producer before being read. Dead-write detection is achieved via a logical-register-indexed table known as the 'Operand Rename Table' (ORT) and is similar to a register alias table / rename map table. When a dynamic instruction's confidence counter saturates, the IR-predictor signals the A-stream to remove that instruction. The dead-write criterion not only applies to original dead writes, but also iteratively to newly dead writes – *as perceived by the A-stream* – by virtue of having removed branches, their direct producers, and so on, in the A-stream. That is, the dead-write criterion iteratively removes branches' backward slices by not considering predicted-removed A-stream instructions as readers of a register. The IR-predictor was augmented to skip fetching of entire basic blocks if they are confidently removable as a whole, thereby amplifying both fetch and execution width of the A-stream core [Pur00]. The Delay Buffer streams a complete control-flow history (confi-

dent predictions for removed branches and resolved outcomes for non-removed branches) and a partial data-flow history from the A-stream to the R-stream, the latter injecting the outcomes as branch predictions in the fetch stage and value predictions in the rename stage. A branch or value misprediction in the R-stream constitutes an instruction-removal misprediction (IR-misprediction) in the A-stream. The A-stream is recovered from the R-stream’s architectural register state. For the slipstream memory hierarchy, the A-stream can read and write its private L1 data cache but evicted dirty blocks are simply discarded as the A-stream is in any case speculative; moreover, a later miss to the discarded block may happen to get the updated block from the trailing R-stream (writes through to shared L2 cache) [Pur02]. Recovery of the A-stream’s L1 data cache was studied in depth [Pur02]; a heuristic of invalidating only dirty blocks yielded performance close to ideal recovery without invalidating the whole cache.

3.2 Slipstream 2.0 Microarchitecture

At a high level, Slipstream 2.0 follows the same microarchitectural blueprint as original Slipstream, so we use similar terms for the added top-level components. The microarchitecture is shown in Figure 3.2.

The A-stream and R-stream run on two cores. Each core has private L1 instruction and data caches. The L2 and L3 caches are shared between the cores.

The added units and modifications for both Slipstream and Slipstream 2.0 are:

- The A-stream’s L1 data cache is speculative, hence, it discards evicted dirty blocks rather than write them back [Pur02]. If the A-stream is squashed and restarted from the R-stream, we follow the invalidate-dirty-block policy for approximately rolling back the A-stream’s L1 data cache [Pur02].
- Both use a **Delay Buffer** to communicate pre-executed outcomes from the A-stream to the R-stream. Whereas Slipstream communicated both branch and value outcomes,

Figure 3.2 Slipstream Processor 2.0 Microarchitecture. Shaded components and dotted lines represent the newly added structures and their connectivity

Slipstream 2.0 only communicates branch outcomes.

- **Instruction-Removal Detector (IR-detector):** This is the unit that monitors the retired instruction stream and uses criteria to identify instructions that should be removed in future instances. The IR-detector trains the IR-predictor (next bullet) which performs the actual instruction removal at the superscalar's instruction fetch stage. As explained in chapter 1, Slipstream 2.0 introduces a wholly new IR-detector, with criteria based on delinquent branches, delinquent loads, and their branch sets.
- **Instruction-Removal Predictor (IR-predictor):** This is the unit that prunes instructions at the instruction fetch stage. Whereas Slipstream used per-dynamic-instance confidence counters for pruning arbitrary instructions in a context-sensitive (global branch history) manner, Slipstream 2.0 uses a much simpler and smaller IR-predictor.

It maintains an entry for each static delinquent pre-executable branch, static delinquent load, and static branches in their branch sets.

In the remainder of this section, we focus on the units with new implementations that distinguish Slipstream 2.0: IR-detector (Section 3.2.1), IR-predictor (Section 3.2.2), and Delay Buffer (Section 3.2.3).

3.2.1 IR-detector

Figure 3.3 shows the three subcomponents of the IR-detector. The boxes labeled “Identify Delinquent Branches/Loads” and “Identify Reconvergent Points” are independent support mechanisms that supply information needed by “Branch Set Analysis”.

Figure 3.3 IR-detector.

3.2.1.1 Identify Reconvergent Points

This subcomponent is continuously training the reconvergent points of branches as they retire, independent of the other subcomponents. We adapted a reconvergence predictor

from the literature [Col04].

The one-entry Active Reconvergence Table (ART) infers the reconvergent point of one static branch at a time over multiple dynamic instances of the branch. It examines program counters (PCs) of instructions retired after the branch and compares them against the same information from past instances, analogous to maintaining a high water mark. These ongoing comparisons gradually increase confidence in the inferred reconvergent point. The *Conf.* field of the ART is incremented after each instance of the branch. When it saturates, the reconvergent point is updated for that branch in the Reconvergence Predictor Table (RPT), and it moves on to analyzing a different branch that comes along.

The RPT holds potential reconvergent points of all branches. We adapted the RPT to hold up to three different reconvergent points per branch. Each has a confidence counter that is initialized or incremented when that reconvergent point is first added or updated, respectively, by the ART. When another subcomponent needs to reference the RPT to get a reconvergent point prediction, the RPT returns the one with the highest confidence. We found that this feature is important to filter-out a distant reconvergent point that may be “truer” in static terms but infrequently exercised in dynamic terms.

The reconvergence predictor (including ART and RPT) requires 1.1 KB of storage, as shown in Figure 3.3.

3.2.1.2 Identify Delinquent Branches/Loads

Delinquent branches and loads are identified by the Branch/Load Classifier (BLC) shown in Figure 3.3. The BLC is indexed and tagged by PC. Each entry has a bit indicating whether it is a branch or load. Each entry also has a 16-bit misprediction/miss counter.

The BLC operates on an epoch basis where an epoch is 500K cycles. All counters are cleared at the beginning of an epoch. A branch’s or load’s counter is incremented each time it is mispredicted or misses in the L2 cache, respectively.

A separate smaller table called “BLC-Max” incrementally maintains a list of the top-8

delinquent loads and branches up to the current point in the epoch. When the BLC is updated, BLC-Max is searched to see if that branch/load is already in the top-8 (matching BLC index found, so just update its counter in BLC-Max) or should knock-off one of the current top-8 (its BLC index does not match any in BLC-Max but its counter is greater than the least counter in BLC-Max). By incrementally maintaining the top-8 list, we avoid serially scanning the BLC for the top 8 at the end of the epoch.

At the end of the epoch, the top-8 delinquent branches and/or loads are sent to the third subcomponent for Branch Set Analysis. The information supplied to Branch Set Analysis is the PC of the branch or load, whether it is a branch or load, and a reconvergent-PC if it is a branch. Thus, at the end of an epoch, the BLC and RPT supply information to kick-off Branch Set Analysis in the next epoch for up to 8 branches/loads. Note that these 8 branches/loads are queued in the unit that does Branch Set Analysis because the analysis is done for one branch or load at a time. After the BLC and RPT kick-off Branch Set Analysis for the next epoch, they return to doing their thing autonomously in the next epoch: the BLC clears all of its misprediction/miss counters and begins anew and the RPT continues training reconvergent points as before.

A 128-entry BLC was found to be sufficient to capture most delinquent loads and branches. The least delinquent branch/load is replaced when there is contention for space in the BLC. Altogether, the BLC and BLC-Max combine for 0.8KB of storage, as shown in Figure 3.3.

Table 3.1 Branch Load Classifier - (49 bits x 128 entries) + (23 bits x 8 entries)

PC	Branch/Load	Prediction State (00: Profiling, 01: DBP, 10: CIDD, 11 DLP)	Mispred/Miss Count
30 Bits	1 Bit	2 Bits	16 Bits

BLC index	Count
7 Bits	16 Bits

3.2.1.3 Branch Set Analysis

As shown in Figure 3.3, the Branch Set Buffer (BSB) learns the branch set for one branch or load at a time.

First, the BSB dequeues the next branch or load from its top-8 queue (the queue is not explicitly shown). It writes the branch/load PC and branch reconvergent-PC (for branches only) into the first two fields. In this paper, up to 32 CIDD branches are identified (could do with much fewer in general). Thus, there are 32 CIDD fields in the BSB, each comprised of a valid bit and PC; the valid bits are not explicitly shown but they are initialized to 0. The control-independent data-independent (CIDI) bit (field labeled “CIDI branch”) is initialized to 1 for a delinquent branch starting out branch set analysis. This bit remains 1 for as long as the delinquent branch is not itself added as a CIDD branch (self-dependent). If it is detected as a CIDD branch with respect to itself, however, the CIDI bit is cleared. At the end of branch set analysis, a delinquent branch with CIDI bit of 1 is deemed pre-executable.

Second, the BSB begins the process of identifying CIDD branches with respect to the delinquent branch/load. This is facilitated by forward poisoning of logical registers influenced by the branch or load, using the Data Dependence Tracker (DDT). The DDT is indexed by logical register specifier (r0-r63 for RISC-V ISA) for source/destination registers. Each DDT entry is a single poison bit indicating whether or not that logical register was directly or transitively influenced by the most recent instance of the delinquent branch/load.

Each time the delinquent branch/load is retired (as known by PC field in the BSB), the DDT is flash-cleared. For a delinquent branch, each retired instruction in its CD region (as known by reconvergent-PC in the BSB) sets the poison bit of its logical destination register if it has one. That is, any register modified in the CD region is poisoned. After reconvergence, each retired control-independent instruction propagates poison bits from any of its logical source registers to any of its logical destination registers. This aspect identifies CIDD instructions. If a retired control-independent branch has any poisoned source registers, it is added to the CIDD branch list in the BSB if it is not already listed. Moreover, if this CIDD branch is the same as the delinquent branch being analyzed (PC match), the BSB clears the CIDI bit of the delinquent branch (not pre-executable). Finally, if a CIDD branch is detected, the poisoning process temporarily reverts to poisoning all logical destination registers in the CD region of the CIDD branch. This ensures transitive-CIDD branches will also be detected, where a transitive-CIDD branch does not directly depend on the delinquent branch but does depend on one of its other CIDD branches. The overall process is the same for a delinquent load, with the exception that it does not have its own CD region to poison (just CD regions of its CIDD branches as just explained).

Poisoning is restarted fresh (DDT flash-cleared) at each retired instance of the delinquent branch/load in the BSB. The process repeats for multiple instances so that as many paths as possible are explored. The final field in the BSB, labeled “Conf.”, is a counter that is incremented for each instance analyzed. When it saturates, the BSB deems the CIDD branch list and the CIDI bit to be accurate.

The final step is to train the IR-predictor when this saturation point is reached. Each IR-predictor entry corresponds to a single branch or load. If it is a branch, it is also annotated as CIDI or CIDD. The IR-predictor is updated by the BSB as follows. The delinquent branch/load is added to (if not already present) or updated in (if already present) the IR-predictor. If it is a delinquent branch, its CIDI bit in the IR-predictor is updated according to its CIDI bit in the BSB. Then, all valid CIDD branches in the BSB (the branch set) are

also added to or updated in the IR-predictor; naturally, their CIDI bits are cleared in the IR-predictor.

Altogether, the BSB and DDT have a storage cost of 0.1 KB.

Table 3.2 Branch Set Buffer - 1025 bits x 1 entry = 128B

Delinquent PC	Reconvergence PC	Branch 1 PC	Branch 2 PC	..29 Branch PCs..	Branch 32 PC	CIDI Branch	Conf
30 Bits	30 Bits	30 Bits	30 Bits	..30 Bits x 29..	30 Bits	1 Bit	4 Bits

Table 3.3 Data Dependence Tracker - 64 Bits Total

R0 Poisoned	R1 Poisoned	R2 Poisoned		R63 Poisoned
1 Bit	1 Bit	1Bit	..1 Bit x 60..	1 Bit

3.2.2 IR-predictor

Table 3.4 shows the fields of an entry in the IR-predictor. The A-stream's instruction fetch unit indexes the IR-predictor by PC.

If there is a hit on a load entry, the fetch unit converts the load to a non-binding prefetch.

If there is a hit on a CIDI branch entry, i.e., “Pre-execute bit” equal to 1, the fetch unit marks the branch for pre-execution and redirects the fetch PC to its reconvergent-PC. Marking it for pre-execution means resolution of the branch one way or the other does not cause a squash in the A-stream.

Finally, if there is a hit on a CIDD branch entry, i.e., “Pre-execute bit” equal to 0, the fetch unit discards the branch instruction and redirects the fetch PC to its reconvergent-PC.

The final field labeled “Mispred/Miss Count” is used to guide replacement of the least-delinquent branch/load when the IR-predictor is using all entries.

We used a 128-entry IR-predictor, which has a storage cost of 1.2 KB.

Table 3.4 IR Predictor Entry - 79 bits x 128 entries

Valid	PC	Branch/Load	Pre-execute bit	Reconvergence PC	Mispred/Miss Count
1 Bit	30 Bits	1 Bit	1 Bit	30 Bits	16 Bits

3.2.3 Delay Buffer

Outcomes of A-stream-executed branches are passed to the R-stream through the Delay Buffer. These outcomes are used by the R-stream to predict its corresponding branches, overriding its branch predictor. If a Delay Buffer outcome turns out to be wrong, the R-stream squashes and restarts the A-stream.

Each entry in the Delay Buffer is 3 bits. The first bit indicates whether this branch was executed (1) or discarded (0) by the A-stream. The second bit indicates the outcome, if executed. The third bit indicates whether (1) or not (0) this branch’s CD region was skipped in the A-stream. Assuming the first bit is most-significant:

- A branch that is executed by the A-stream normally along with its CD region is encoded as 1x0 (x is outcome). The R-stream can use the outcome, moreover, subsequent outcomes line up, too (no gap in Delay Buffer).
- A pre-executed branch is encoded as 1x1 (x is outcome). The R-stream can use the outcome, but it knows there may be a gap in the Delay Buffer after this branch until its reconvergent-PC is fetched. Thus, the R-stream reverts to its branch predictor for

any branches (if any) that are fetched before the reconvergent-PC is fetched.

- A branch that was discarded by the A-stream, because it is CIDD, is encoded as 0-1. The R-stream reverts to its branch predictor for the A-stream-discarded branch because there isn't a valid outcome for it. It also knows there may be a gap in the Delay Buffer after this branch until its reconvergent-PC is fetched. Thus, the R-stream continues using its branch predictor for any branches (if any) that are fetched before the reconvergent-PC is fetched.

We used a 256-entry Delay Buffer in this paper. This translates to just 0.1KB if we assume the method discussed above, which assumes the R-stream can access the RPT directly for reconvergent-PCs. Another strategy is to include a reconvergent-PC in each Delay Buffer entry, pushed by the A-stream when the third bit is 1. This translates to 1KB. In the next section, we assume the higher storage cost for the Delay Buffer.

3.2.4 Storage cost

The total storage cost for the IR-detector, IR-predictor, and Delay Buffer, is 4.2KB.

- IR-detector: 2KB (0.8KB for BLC and BLC-Max; 1.1KB for ART and RPT; 0.1KB for BSB and DDT)
- IR-predictor: 1.2KB
- Delay Buffer (if each entry includes a reconvergent-PC so that R-stream need not access the RPT): 1KB

CHAPTER

4

CASE STUDIES

The fundamental ways in which DBP and DLP captures the opportunities presented by certain program characteristics are discussed below with actual code examples.

4.1 Delinquent Branch Pre-execution

For a delinquent branch that is not data dependent on its CD regions, also known as a separable branch (partially separable branch when it is data independent most of the time), the branch is executed but its CD region is removed from execution. The branch outcomes are fed to the R-stream thus speeding it up.

4.1.1 Case Study: ASTAR

To demonstrate delinquent Branch Pre-execution, integer benchmark "astar" from SPEC 2006 is used as an example. Specifically, the makebound2 function is chosen as it takes up 48% of the total execution time. The region of interest is a loop as shown in listing 4.1:

```
1 for (i=0; i<bound1l; i++)
2 {
3   index=bound1p[ i ];
4
5   index1=index-yoffset-1;
6   if (waymap[ index1 ]. fillnum!= fillnum )
7     if (maparp[ index1 ]==0)
8     {
9       bound2p[ bound2l]=index1 ;
10      bound2l++;
11
12      waymap[ index1 ]. fillnum=fillnum ;
13      waymap[ index1 ]. num=step ;
14
15      if (index1==endindex)
16      {
17        flend=true ;
18        return bound2l;
19      }
20    }
21 }
```

Listing 4.1 astar makebound2 function

This loop iterates over an input worklist 'bound1p' and generates an output worklist 'bound2p'. The code has been shortened for brevity. In the original code, line 6-20 is replicated 8 times for each neighboring cell. The nested if branches in lines 6 and 7 are load-dependent and often mispredict. Often times, the load feeding the branches miss in

the caches. These branch mispredictions are responsible for the high execution time of this loop. But this branch is a partially-separable branch as the only dependency between the branch and its CD region are the load-store synchronization in lines 6 and 12. But this dependency doesn't exist on most iterations. Running this benchmark with DBP, we observe that the delinquent branches in lines 6 and 7 are pre-executed but its CD region is not executed. Thus we can pre-execute all such delinquent branches in this loop. Whenever the load-store synchronization is violated, it results in a wrong branch outcome from the A-stream. This causes a branch misprediction in the R-stream which then recovers both R-stream and A-stream. But whenever it is safe to violate the synchronization, A-stream can pre-execute these branches and produce accurate branch outcomes for the R-stream. For the branches inside the CD region, the R-stream uses its own branch predictor to predict them. After that, it resumes back to using the delay buffer after the reconvergent point. There can be cases where the CD region branches cause both the streams to go out of sync with each other. In cases like that, the delay buffer fills up but no useful prediction is available for the R-stream. The R-stream detects this case and restarts the A-stream accordingly.

4.1.2 Case Study: hmmer

The P7Viterbi function in hmmer has the top execution time and has a lot of delinquent branches in it. The branches are shown in listing 4.2.

The loop iterates over source arrays and creates output arrays `mc[k]`, `ic[k]`. The code is shortened to show only the highly mispredicting branches of concern. The high MPKI branches are in lines 4 and 11. The CD region of these branches is part of the backward slice of the branches in 5 and 13. Hence branches in line 5 and 13 are part of the branch set of the delinquent branches 4 and 11. However, the branches in lines 3-5 are independent of the branches in lines 11-13. And the delinquent branches themselves are not in its own branch sets. Hence applying DBP on this loop we can pre-execute the branches in lines 4 and 11

and skip the branches in its branch set. The next instance of branches 4 and 11 are not impacted because of skipping their branch sets. Exploiting this insight about delinquent branches and its relation to other branches helps DBP achieve more performance.

```

1 for (k = 1; k <= M; k++) {
2   mc[k] = mpp[k-1]+tpmm[k-1];
3   if ((sc = ip[k-1]+tpim[k-1]) > mc[k]) mc[k] = sc;
4   if ((sc = dpp[k-1]+tpdm[k-1]) > mc[k]) mc[k] = sc;
5   if ((sc = xmb+bp[k]) > mc[k]) mc[k] = sc;
6   mc[k] += ms[k];
7   if (mc[k] < -INFTY) mc[k] = -INFTY;
8
9   if (k < M) {
10    ic[k] = mpp[k]+tpmi[k];
11    if ((sc = ip[k]+tpii[k]) > ic[k]) ic[k] = sc;
12    ic[k] += is[k];
13    if (ic[k] < -INFTY) ic[k] = -INFTY;
14  }
15}

```

Listing 4.2 hmmer P7Viterbi function

4.2 Delinquent Load Prefetch

For applications with high cache MPKI (Misses per 1K Instructions), the A-stream is slowed down as it waits for the cache misses to resolve to make forward progress. As a result, the A-stream is not sped up enough to provide timely branch outcomes to the R-stream. In Delinquent Load prefetch, these delinquent loads are identified and converted into prefetches in the A-stream to generate these misses in parallel. The load-dependent branches are identified and are removed from execution by the IR-detector. Thus A-stream is not impacted by load data-dependent branches. R-stream executes all such branches using its own branch

predictor but now has prefetched cache blocks in its L2 caches. DLP benefits applications like libquantum that has many delinquent loads that often miss in the caches.

4.2.1 Case Study: libquantum

```

1 for (i=0; i<reg->size; i++)
2 {
3   / Flip the target bit of a basis state if both control bits are
      set /
4   if (reg->node[i].state & ((MAX_UNSIGNED) 1 << control1))
5   {
6     if (reg->node[i].state & ((MAX_UNSIGNED) 1 << control2))
7     {
8       reg->node[i].state ^= ((MAX_UNSIGNED) 1 << target);
9     }
10  }
11 }

```

Listing 4.3 libquantum quantum_toffoli function

A code snippet from libquantum's quantum_toffoli function is shown in listing 4.3 to demonstrate how DLP works. This function iterates over the 'reg' array and sets a target bit if both control bits are set. Even though the branches themselves are data-dependent, they are highly biased and are easy to predict. But the load to the state variable in line 4 often misses in the L2 and takes hundreds of cycles to resolve. In spite of the load addresses following a prefetchable pattern, even an aggressive hardware prefetcher cannot completely hide the latency. The application ends up with a fairly high cache Average Access Time(AAT). While running with DLP, in quantum_toffoli the load in line 4 is converted to prefetch. It's dependent branches in lines 6 and 8 are removed from execution as they are part of the branch set of load in line 4. The branches in line 6 and 8 are data-dependent and can lead to incorrect execution outcomes. These branches are not executed in the A-stream. However,

these are highly predictable and R-stream's predictor does a very good job of speculating past them.

4.2.2 Case Study: mcf

```

1 for( ; arc < stop_arcs; arc += nr_group )
2 {
3   if( arc->ident > BASIC )
4   {
5     red_cost = arc->cost - arc->tail->potential + arc->head->
        potential;
6     if((red_cost < 0 && arc->ident == AT_LOWER) ||
7       (red_cost > 0 && arc->ident == AT_UPPER))
8     {
9       basket_size++;
10      perm[basket_size]->a = arc;
11      perm[basket_size]->cost = red_cost;
12      perm[basket_size]->abs_cost = ABS(red_cost);
13    }
14  }
15 }

```

Listing 4.4 mcf primal_bea_mpp function

Another example where DLP works well is a hot function in mcf, primal_bea_mpp, which has the top execution time. This function also has long latency loads feeding data dependent branches. However contrary to quantum_toffoli these branches are hard to predict. A particular region of the function is shown in listing 4.4.

The loads in lines 3 and 5 are all delinquent loads and often miss in its cache. The loads then feed the branches in lines 6 and 7 which mispredict often. Applying DLP on mcf the delinquent loads in lines 3 and 5 are converted to prefetches and are early retired. The load dependent branch in line 7 is identified by the branch set analysis is skipped in the A-stream and executed in the R-stream. Without the branch misprediction bottleneck, A-stream

can now run faster. This way A-stream acts as the prefetcher and it also helps resolve the branches in the R-stream faster.

CHAPTER

5

RECONVERGENCE PREDICTOR

The reconvergence point of a branch is the instruction where all the diverging control flow paths from the branch eventually meet and continue on to a straight line code stream. Any instruction in the dynamic instruction stream before the reconvergence point and after the branch itself is said to be control dependent on the branch. Any instruction after the reconvergence point is control independent. A reconvergent predictor that keeps track of instructions that retire after the branch can predict what the reconvergent point of a branch is dynamically with good accuracy. This work utilizes one such reconvergent predictor that is similar to the one mentioned here [Col04]. The tables used in the reconvergence predictor are shown in Table 5.1.

The reconvergence predictor table (RPT) entries are shown in Table 5.1a. It stores the potential reconvergence points of all branches. As some branches could have more than

Table 5.1 Formats of the various tables in the Reconvergence Predictor.

(a) Reconvergence Predictor Table - 140 Bits Per Entry x 64 entries

Branch PC	Potential Reconvergence PC	Confidence	Potential Reconvergence PC	Confidence	Potential Reconvergence PC	Confidence
30 Bits	30 Bits	4 Bits	30 Bits	4 Bits	30 Bits	4 Bits

(b) Active Reconvergence Table - 102 Bits Per Entry x 1 entry

Active Branch PC	Active Below Potential PC	Active BP	Active Above Potential PC	Active AP	Confidence
30 Bits	30 Bits	1 Bit	30 Bits	1 Bit	4 Bits

one reconvergent point, the RPT can keep track of a total of 3 reconvergent points and a confidence estimation mechanism determines the most frequently taken reconvergent point for this branch. This is important for branches that have a rarely taken return inside their CD region that takes control out of the loop to a caller function. In cases like these, the reconvergence predictor estimates how often the return is taken and predicts the more commonly executed instruction as the reconvergent point that is most likely the target or an instruction that is closer to the target.

The RPT is fed by the Active Reconvergence Table (ART) whose entries are shown in Table 5.1b. The ART is set with the current active branch and:

- i. The Active Below potential PC is updated to be the PC of the instruction immediately after the branch. And the active BP is set.
- ii. If the Below potential PC is retired after the branch the active BP is cleared. If a PC that is higher than below potential executes, then that PC is set as the new below potential and Active BP is cleared.
- iii. In case, a PC that is lesser than the branch PC executes after the branch, then it is a

backward branch and it updates the Above potential PC and sets the Active AP bit.

- iv. When the branch executes again it increments the confidence of the branch as well as allocates the Below/Above Potential PC to a Potential Reconvergent PC entry for the active branch in the RPT. It also increments the confidence of the reconvergent point indicating how many times a potential reconvergent point has been observed.
- v. After the current active branch has been seen a sufficient number of times and its reconvergent PC's are all collected in the RPT, the reconvergence predictor moves on to the next branch and sets it as the currently active branch.

During prediction, the RPT is looked up with the branch PC and the most confident PC of all potential reconvergent PC is chosen as the predicted reconvergent point. Even though we collect reconvergent points for backward branches, we do not skip these branches as it could create deadlocks in the A-stream. The reconvergence predictor used in our design requires around 1.1 KB of storage.

5.1 Reconvergence Predictor Example

In figure 5.1, initially the RPT and ART is set with Br0, the branch we want to find the reconvergence point for, the active below potential PC is set to the next PC to the branch. In this case, it is instruction i0 and the active BP bit is set to indicate that we are monitoring the instruction stream for i0 or PC that is lower than that. As Br0 executes not taken, i0 executes, active BP bit is cleared and confidence is incremented to indicate i0 could be a potential reconvergence point for Br0. The next instance of Br0 we again set i0 as the active below potential PC.

Reconvergence Predictor Table						
Branch PC	Potential Reconvergence PC 1	Confidence	Potential Reconvergence PC 2	Confidence	Potential Reconvergence PC 3	Confidence
br0	i0	0000				

Active Reconvergence Table					
Active Branch PC	Active Below Potential PC	Active BP	Active Above potential PC	Active AP	Confidence
br0	i0	1 bit			0000

Reconvergence Predictor Table						
Branch PC	Potential Reconvergence PC 1	Confidence	Potential Reconvergence PC 2	Confidence	Potential Reconvergence PC 3	Confidence
br0	i0	0001				

Active Reconvergence Table					
Active Branch PC	Active Below Potential PC	Active BP	Active Above potential PC	Active AP	Confidence
br0	i0	0 bit			0001

Reconvergence Predictor Table						
Branch PC	Potential Reconvergence PC 1	Confidence	Potential Reconvergence PC 2	Confidence	Potential Reconvergence PC 3	Confidence
br0	i0	0001				

Active Reconvergence Table					
Active Branch PC	Active Below Potential PC	Active BP	Active Above potential PC	Active AP	Confidence
br0	i0	1 bit			0001

Figure 5.1 On-going Reconvergence Predictor example (snapshot 1).

But this time, as shown in figure 5.2, Br0 executes taken. Instruction i1 executes after the branch and since I1 is higher than I0, the active below potential PC is updated to I1 and active BP bit is cleared. I1 is also allocated a new entry in the RPT and it's confidence is incremented to indicate I1 could be a potential reconvergent point. In the next iteration, we

Figure 5.2 On-going Reconvergence Predictor example (snapshot 2).

set I1 as the active below potential PC since it is the highest potential reconvergent point that we have seen so far and active BP bit is set. This time Br0 executes not taken again, hence I1 doesn't execute, i0 executes. Since I0 is lower than I1 there is no updates to the ART. But as control flow merges back, instruction i2 is executed, and now since i2 is higher than

i0 or i1, it is updated as the new active below potential PC and active BP bit is cleared. We repeat this process for many instances of Br0 and observe all potential reconvergent PCs.

Figure 5.3 On-going Reconvergence Predictor example (snapshot 3, final).

The final state of the reconvergence predictor is shown in figure 5.3. After 16 instances of Br0, as it can be seen from the confidence counter, we have observed i2 as the potential reconvergent PC for 14 such instances. Since i2 has the maximum confidence in the RPT when Br0 indexes the RPT, i2 is predicted as its reconvergent point by the reconvergence predictor.

CHAPTER

6

BRANCH SET ANALYSIS EXAMPLE

The analysis in figure 6.1 through figure 6.6 shows the control flow graph (CFG) of an application. The control-flow path is highlighted to show program behavior. The BLC identifies Br1 to be a delinquent branch, and it sent to the BSB for branch set analysis. The reconvergence predictor supplies the reconvergent point of the branch and it is used by the DDT. The DDT tracks logical registers that are written in the control dependent region of the branch by poisoning them. All destination registers that are written between the branch and its reconvergent points are poisoned in the DDT. DDT then propagates the poison bit through instructions by also poisoning the destination register of the instruction that uses a poisoned register as a source. If a register is written by an instruction that not in the CD region of the branch, the poison bit is cleared to indicate a new writer. The DDT is reset every time the delinquent branch is encountered and the analysis restarts.

Branch Set Buffer			
Branch PC	CIDI-branch	Branch Set	Confidence
Br1	false	<>	01

DDT	
R0	0
R1	0
R2	0
R3	0
R4	0
R5	0

Branch Set Buffer			
Branch PC	CIDI-branch	Branch Set	Confidence
Br1	false	<>	01

DDT	
R0	0
R1	0
R2	0
R3	0
R4	1
R5	0

Figure 6.1 Branch Set Analysis example (snapshot 1).

In our example CFG, first, for Br1 all instructions that are in the control dependent (CD) region of the branch is identified using the reconvergence point of the branch. Then for each CD instruction retiring, their destination logical register is marked as "poisoned" in the DDT. As R4 is produced in the CD region of Br1 it is marked as poisoned in the DDT. If any of the instructions consumes a poisoned register they pass the poisoned register to its consumers. If a branch consumes a poisoned source value after the reconvergence point then it is a control-independent data-dependent (CIDD) branch to the delinquent branch.

Branch Set Buffer			
Branch PC	CIDI-branch	Branch Set	Confidence
Br1	false	<Br4>	01

DDT	
R0	0
R1	0
R2	0
R3	0
R4	1
R5	0

Branch Set Buffer			
Branch PC	CIDI-branch	Branch Set	Confidence
Br1	false	<Br4>	01

DDT	
R0	0
R1	0
R2	0
R3	0
R4	1
R5	1

Figure 6.2 Branch Set Analysis example (snapshot 2).

Hence it is added to the branch set of the current delinquent branch we are analyzing. In Figure 6.2, R4 is consumed by Br4 so it is added to the branch set of Br1.

Whenever we add a branch to the branch set we obtain the reconvergent point of that branch and then poison all destination registers that are in the CD region of that branch as well. This is done in order to ensure the poisoning is done transitively to all branches in a branch's branch set. Now r5 is poisoned as it is in Br4's CD region. Br7 utilizes r5 which is written inside the CD region of Br4, hence it is added to the branch set of Br1 as well.

Branch Set Buffer			
Branch PC	CIDI-branch	Branch Set	Confidence
Br1	false	<Br4, Br7>	01

DDT	
R0	0
R1	0
R2	0
R3	0
R4	1
R5	1

Branch Set Buffer			
Branch PC	CIDI-branch	Branch Set	Confidence
Br1	false	<Br4, Br7>	01

DDT	
R0	0
R1	0
R2	0
R3	0
R4	1
R5	1

Figure 6.3 Branch Set Analysis example (snapshot 3).

Now the first iteration is complete. And when next instance of Br1 is encountered, the confidence counter is incremented to indicate we have analyzed at least one instance of Br1 and its dependent branches. During the second iteration the control-flow changes.

Branch Set Buffer			
Branch PC	CIDI-branch	Branch Set	Confidence
Br1	false	<Br4, Br7>	10

DDT	
R0	0
R1	0
R2	0
R3	0
R4	0
R5	0

Branch Set Buffer			
Branch PC	CIDI-branch	Branch Set	Confidence
Br1	false	<Br4, Br7>	10

DDT	
R0	0
R1	0
R2	0
R3	0
R4	0
R5	1

Figure 6.4 Branch Set Analysis example (snapshot 4).

Now on this path r5 is poisoned as it is written in the CD region. Since Br4 doesn't use r5 it is not poisoned. On the other hand, Br7 consumes the poisoned value of R5 should be added to the branch set. Since it is already added it is not added again.

Branch Set Buffer			
Branch PC	CIDI-branch	Branch Set	Confidence
Br1	false	<Br4, Br7>	10

DDT	
R0	0
R1	0
R2	0
R3	0
R4	0
R5	1

Branch Set Buffer			
Branch PC	CIDI-branch	Branch Set	Confidence
Br1	false	<Br4, Br7>	10

DDT	
R0	0
R1	0
R2	0
R3	0
R4	0
R5	1

Figure 6.5 Branch Set Analysis example (snapshot 5).

The same process is repeated on the next instance of Br1, until no more new branches are added to the branch set of Br1. As the confidence reaches a threshold we have now captured all CIDD branches of Br1 in its branch set.

Figure 6.6 Branch Set Analysis example (snapshot 6, final).

The final state of the tables and the A-stream formed is shown in figure 6.6. Since Br1 itself is not in its own branch set it is determined as control-independent and data-independent (CIDI) with respect to itself. Hence it is identified as a branch that is fit for DBP and DBP is enabled in the corresponding IR predictor entry. All the branches in the branch set are also allocated in the IR predictor as they are CIDD of this branch.

For delinquent loads, the analysis is similar to find load dependent branches to add to its branch set. However, delinquent loads are not control flow instructions and don't need a reconvergence point. Instead of branch's CD instructions setting the poison bit in the DDT, delinquent loads poison its own destination register in the DDT and when a branch consumes a poisoned source value it is added to the load's branch set.

CHAPTER

7

EVALUATION

7.1 Benchmark Analysis

The benchmarks used in this study are part of the SPEC 2006 and SPEC 2017 suites. The top 3 Simpoints for each with 100 million instruction regions were generated and analyzed. As part of our design space exploration, we analyzed more than 25 benchmarks with varying branch MPKI or cache MPKI bottleneck.

7.2 Simulator Configuration

We use a detailed, cycle-level, execute-at-execute superscalar core simulator that executes the RISC-V ISA. It can be configured for a single core or for dual cores in Slipstream 2.0,

Slipstream, or Dual Core Execution modes. For all experiments, each superscalar core is configured similarly to the Intel Skylake Desktop Processor (where we have common and major superscalar parameters we set them according to Skylake). This configuration is shown in Table 7.1.

The 22nm technology node provided by McPAT [Li09] was used to calculate energy for Slipstream 2.0 (two cores) and the baseline (one core). New Slipstream 2.0 components were added to McPAT, with their activity factors gotten from the timing simulator as with other components.

Table 7.1 Core configuration with parameters modelled after Intel Skylake Desktop Processor, for in-common major superscalar parameters.

Microarchitecture	Fea- ture	Details
Branch Prediction		BP: 64KB TAGE-SC-L predictor [Sez16] BTB: 4K entries, 4-way set-associative, RAS: 32 entries
Hardware Prefetcher		VLDP [She15]: 5.5 Kb
Memory Hierarchy		L1: split, 32KB each, 8-way set-associative, 4-cycle access latency L2: unified, shared 256KB, 4-way set-associative, 12-cycle access latency L3: unified, shared 8 MB, 16-way set-associative, 42-cycle access latency
DRAM		250-cycle access latency
Fetch/Retire Width		4 instr./cycle
Issue/Execute Width		8 instr./cycle
ROB/IQ/LDQ/STQ		224/100/72/72
Simple/LS/FP-Complex ALUs		4/2/2
Fetch-to-Execute Latency		10 cycle
Physical RF		288
Checkpoints		32, OoO reclamation
Slipstream		Delay buffer size: 256 entries

CHAPTER

8

RESULTS

Results are only presented here for the 8 benchmarks for which Slipstream 2.0 is enabled for almost the entire SimPoint by our automatic turbo boost mechanism. The turbo boost mechanism is presented later, in Section 8.5. Section 8.5 presents results for all 25 benchmarks, including correlation between branch/load MPKI, turbo boost enabling/disabling, and speedups.

8.1 Delinquent Branch Pre-execution (DBP)

8.1.1 Speedup

Delinquent Branch Pre-execution gives significant speedups for benchmarks with high Branch MPKI. bzip2, astar and hmmer all achieve 17% to 28% speedup with DBP as shown in

figure 8.1. As branch mispredictions are the main bottleneck in these applications (see Figure 8.13a), the reduction in MPKI is the main reason for the speedup. For these benchmarks we see a 35-65% reduction in branch MPKI. DBP effectively pre-executes all CIDI branches and delegates the CIDD branches to R-stream branch predictor. Since the hard-to-predict branches are now pre-executed in the A-stream, R-stream suffers mispredictions only for branches in the branch set of these branches. As it uses its branch predictor to resolve these branches, the A-stream is pre-executing its share of branches and their outcomes are readily available in the delay buffer. Performance of DBP is comparable to that with perfect branch prediction for bzip2, hmmer, and mcf, as many of the delinquent branches are now pre-executed by the A-stream. In case of astar, nested if statements prevents A-stream from pre-executing all the delinquent branches. So R-stream is left with its share of delinquent branches to execute. Despite this, we see 28% speedup from A-stream pre-execution.

Figure 8.1 Speedups of DBP over baseline, and baseline with perfect branch prediction.

DBP does not provide speedups for application with high cache AAT. libquantum, omnetpp, and bwaves show no speedup because the A-stream is still slowed down significantly by these cache misses. These benchmarks do not have high branch MPKI to begin with and it is evident as even with perfect branch prediction these benchmarks do not see any

speedup. R-stream waits on A-stream to provide these outcomes hence we see no speedup.

8.1.2 Number of Delinquent and Pre-executable Branches

Figure 8.2 Delinquent branches and CIDI branches observed in DBP.

Figures 8.2a shows how many unique (static) delinquent branch PCs are observed in each benchmark for Slipstream 2.0. Figure 8.2a also shows the number of unique (static) CIDI branches available in each benchmark. CIDI branches are the totally separable branches that are chosen for pre-execution by DBP. The more the number of pre-executable branches, the higher the chances of A-stream speeding up that application without incurring IR mispredicts. Note that the memory bound applications have low to zero delinquent branch count hence no CIDI branches were found for these applications. In bzip2, hmmer, astar, and mcf, branch set analysis was able to find a sufficient number of CIDI branches which creates the main opportunities for pre-execution which DBP successfully exploits. Figure 8.2b additionally shows the percentage of pre-executed branches out of total dynamic branches as observed when running with DBP. We can see that a significant portion of dynamic branches is pre-executable in these applications.

8.1.3 Branch Set Analysis

Branch set analysis finds CIDI branches and CIDD branches for each delinquent branch (and also determines whether or not the delinquent branch is CIDI with respect to itself). Figure 8.3 shows a histogram of branch set sizes. This data is important for the heuristics that drive branch and load pre-execution. From Figure 8.3, it is evident that most of the delinquent branches have fewer than two (and often zero) branches in their branch sets. These branches are ideal for pre-execution, if they are CIDI, as the number of other branches' executions it would impact is less. There are also branches that have 10+ branches in their branch sets in bzip2, hmmer, and mcf. Pre-executing these branches might negatively impact performance as you trade-off the benefit of executing all these branches for pre-executing 1 CIDI branch. Our heuristic considers branch set size of a branch as a parameter to decide when to pre-execute a branch. We have a branch set size threshold of 10 branches above which we don't pre-execute a branch. This also helps reduce contention in our IR-predictor table. Since lbm, bwaves, and libquantum have few or no delinquent branches, their branch set occupancy is very sparse.

Figure 8.3 Delinquent branches' branch set sizes observed in DBP.

8.2 Delinquent Load Prefetching (DLP)

8.2.1 Speedup

Figure 8.4 shows speedups of DLP over the baseline. To gauge speedup potential, Figure 8.4 also shows speedups of the baseline with a perfect data cache. DLP benefits benchmarks suffering primarily from L2/L3 cache misses, which are evident from the speedup of "Perfect DC". Libquantum, lbm, omnetpp, mcf, and bwaves, all see speedups ranging from 13% all

the way to 2.9x speedup for libquantum. Mcf exhibits delinquent loads with long average access time (AAT) that also feed mispredicted branches. Slipstream 2.0's ability to do DLP, while insulating the A-stream from miss-dependent mispredictions that resolve in the R-stream (i.e., A-stream is not restarted), is quite beneficial for mcf. This effect is also evident in benchmarks with a milder delinquent load problem and high branch MPKI: bzip2 and hmmer get 17-22% improvement over baseline with DLP, which is quite close to "Perfect DC" for them. In other miss-bound benchmarks, DLP helps close the gap between baseline and perfect data caches.

Figure 8.4 Speedups of DLP over baseline, and baseline with perfect data caches.

8.2.2 Number of Delinquent Loads

Figure 8.5 shows how many unique (static) delinquent load PCs are observed in the benchmarks for Slipstream 2.0. It shows a good number of delinquent loads that were found by the Branch Load Classifier by monitoring the L2 and LLC cache miss rates of benchmarks. These high number of delinquent loads creates opportunities for DLP. Even though the number of static loads found in libquantum is lesser, the loads often miss in the L2 and have high cache AAT. Lbm and mcf have the most static loads that are delinquent and DLP does

its best by capturing these loads and prefetching them. Control-bound applications like bzip2, hmmer, and astar also have good amounts of delinquent loads. Despite the cache AAT of these applications overall being not that high, the criticality of loads that do miss may be amplified if they feed mispredicted branches: DLP is able to fully insulate the A-stream from these cache misses by converting the loads to non-binding prefetches and not rolling back the A-stream if their dependent branches are found out to be mispredictions in the R-stream.

Figure 8.5 Delinquent Loads observed in DLP.

8.2.3 Branch Set Analysis

Figure 8.6 shows branch set sizes for delinquent loads. These are load dependent branches that are supposed to be removed after the load has been converted to a prefetch. The branch set sizes are small for many of the loads, indicating that there are fewer load depen-

dent branches for these applications. If the prediction accuracy for these data-dependent branches is low then they would cause frequent squashes of the A-stream for a microarchitecture like DCE. DLP avoids this problem by removing the entire load's forward control-flow slice when we pseudo-retire that load. Accordingly, the A-stream need not be rolled back when the R-stream detects a load-dependent branch misprediction.

Figure 8.6 Delinquent loads' branch set sizes observed in DLP.

8.3 Delinquent Load Prefetching (DLP) + Delinquent Branch Pre-execution (DBP)

8.3.1 DBP+DLP Proactive

Data-dependent branches, i.e., load-dependent branches, tend to be the primary source of mispredictions. If a load-dependent branch is mispredicted often and if its producer load misses often, both will be classed as delinquent. When combining DBP and DLP we are forced to choose whether to pre-execute the branch or prefetch the load. As once the load is converted to prefetch, the branch will not be able to pre-execute. In cases like these, DBP+DLP proactive removes all the load-dependent slices from execution, at the fetch stage of the pipeline, and ensures we at least get the benefit of the prefetched load. It resorts to choosing DLP over DBP for all benchmarks. This helps us achieve similar speedups as in DLP for most of the applications as seen in Figure 8.7. It is around 59% geometric mean speedup compared to the baseline. We see some performance benefit from DBP for branches that are not dependent on a delinquent load but it is very limited. In applications like *astar*, since pre-executing branches gave most of the performance impact, choosing load prefetch over branch pre-execution results in only 5% speedup.

8.3.2 DBP+DLP Reactive

A delinquent load does not miss in all instances; for hits, the delinquent branch can be pre-executed correctly. Thus, the proactive approach leaves performance on the table because it essentially always assumes a miss and the delinquent branch is never pre-executed.

Utilizing this insight, we devise a reactive DLP approach for misprediction-heavy phases, so that DBP is maximized and only forgone when the delinquent load misses. For reactive DLP, the load's forward control-flow slice is removed at retire – and only for a miss – instead of at instruction fetch. Thus, the delinquent branch is pre-executed on a hit and converted to

Figure 8.7 Speedups of DBP+DLP proactive and reactive over baseline, and baseline with perfect branch prediction and data caches.

a discarded CIDD branch at retire on a miss. Note that the branch does not trigger recovery in any case because it is marked for pre-execution and it already proactively removed its forward control-flow slice. Essentially, with reactive DLP, the load-dependent delinquent branch is allowed to begin as “CIDI pre-executable” and either remains that way (load hit) or downgrades to “CIDD discardable” (load miss): either way, the branch’s forward control-flow slice has been proactively removed, and the only dynamic adjustment is at retire in terms of which encoding to push onto the Delay Buffer for the branch.

With this technique, all benchmarks show speedups when DBP and DLP are combined, giving a geometric mean speedup of close to 67%. DBP+DLP gives a speedup of 22% for hmmer and 32% for bzip2. If the load hits in the cache often then the branch pre-executes, if it misses it is prefetched. Hence for control-bound applications combining both techniques can get significant speedups as the load prefetching and hard-to-predict branch pre-execution complement each other nicely. In other benchmarks, like mcf, libquantum, lbm, omnetpp, and bwaves, we see similar performance to that of applying DLP independently. These benchmarks get speedup only because of the technique that it most favors and gets almost no benefit from the other technique.

Astar also benefits by this approach as most of the loads hit in the cache, we see more

branches getting pre-executed in the A-stream. This helps us utilize the DBP component to its full potential without losing any opportunity. Astar sees an overall speedup of 29% over baseline. Applying both the schemes helps us to combine DBP and DLP in a fine-grained manner. But still, we are not able to both prefetch and pre-execute at the same time. Hence we do not see speedups as much as combining perfect branch prediction with perfect data caches.

Figure 8.8 Delinquent Branches and Loads observed in DBP+DLP.

8.3.3 Delinquent Branches and Loads in DBP+DLP

The delinquent loads and branches that are observed when combining DBP and DLP reactively are shown in Figure 8.8. Corresponding branch set sizes are shown in Figure 8.9 (branches' branch sets) and Figure 8.10 (loads' branch sets).

Figure 8.9 Delinquent branches' branch set sizes observed in DBP+DLP reactive.

Figure 8.10 Delinquent loads' branch set sizes observed in DBP+DLP reactive.

8.4 Comparisons with Slipstream 1.0 and DCE

Classic slipstream's instruction removal rate is very low for applications with high branch mispredictions. Astar, bzip2, and hmmer, all show very low removal rates. As a result, the A-stream is slowed down by mispredictions hence it provides no speedup compared to the baseline as seen in Figure 8.11. DBP overcomes this by identifying CIDI branches to pre-execute. Classic slipstream provides 11% to 14% speedup for libquantum, mcf, and omnetpp. Classic slipstream still executes the load-dependent slices but as the branches in these benchmarks are highly confident, we get a good amount of instruction removal from A-stream. Even though some delinquent loads were removed by the A-stream's back-propagation, some loads still execute thus giving us some prefetching effect. But DLP makes sure that all such loads are converted to prefetches without losing any opportunity.

Figure 8.11 Comparisons among Slipstream 1.0, DCE, and Slipstream 2.0 (DBP, DLP, and DBP+DLP).

Dual Core Execution is different from DLP in the sense that the load-dependent branches are still left to execute in the A-stream but these poisoned branches now depend on its branch predictor for its outcome. The problem with this approach is whenever you have load-dependent branches that mispredict a lot it can limit the performance of the front processor. As the front processor's branch outcomes often end up wrong, it is squashed quite often resulting in lost performance opportunity. DLP overcomes this by effectively

Figure 8.12 Speedup of DBP+DLP over baseline, over Slipstream 1.0, and over DCE. Data is categorized into control-intensive, memory-intensive, and all benchmarks.

removing loads' forward control-flow slices from the leading thread's execution stream. This is particularly useful for benchmarks that mispredict a lot. This effect can be seen in Figure 8.11. Benchmarks that rely on correct branch outcomes, like bzip2, mcf, and astar, show 20% to 30% better speedups with DBP+DLP compared to DCE. For applications that do not have many branch mispredictions, such as libquantum, lbm, and omnetpp, DBP+DLP performs within 5% the speedup provided by DCE. Thus DBP+DLP effectively builds upon DCE and achieves better speedups for control-bound applications as well. Overall, DBP+DLP provides a geometric mean speedup of 12% compared to DCE.

8.5 Microarchitectural Turbo Boost

Despite the speedups from DBP and DLP, the A-stream running on the separate core incurs additional power cost that has to be paid. One solution is to turn on the A-stream only when the power budget of the chip allows it. This way, A-stream is used as a microarchitectural turbo boost. The heuristic used to turn on/off the A-stream based on branch and load performance is shown below:

1EN / AStream is enabled /
 2DBPt / delinquent branch MPKI threshold observed in current epoch
 /

```

3DLPt / delinquent load MPKI threshold observed in current epoch /
4
5 if (end of current epoch reached) {
6   if (DBPt || DLPt) {
7     if (!EN) {
8       Stop execution and fork AStream.
9     } // else keep AStream enabled
10  }
11  else {
12    if (EN) {
13      Stop execution and disable AStream.
14    } // else keep AStream disabled
15  }
16}

```

Listing 8.1 Heuristics to turn on/off A-stream

Using this simple heuristic we ran all the benchmarks and the A-stream utilization is shown in Figure 8.13b. 17 of the 25 benchmarks that we analyzed had A-stream disabled for 60% or more of execution time and 8 of the 25 benchmarks had A-stream enabled 100% of execution time. The 8 applications that always use the A-stream show speedups as shown in Figure 8.13c. 10 of the 17 benchmarks that did not use the A-stream have branch misprediction rates and cache miss rates less than the delinquency thresholds as shown in Figure 8.13a. These benchmarks could either have high IPC or be bound by true data dependencies. 3 of the remaining 7 benchmarks had a delinquent branch and/or load problem only during certain phases of the application and hence enabled the A-stream for 10%-40% of execution time and provided no overall speedup. 4 of the remaining 7 are perlbench, sjeng, leela_s, and wrf_s. All of these applications either had a lot of non-pre-executable branches or the pre-executable branches that were removed were not delinquent enough to provide a speedup to the R-stream. Since the number of DBP branches and their delinquency are input parameters to the heuristic above it results

in lesser utilization of A-stream for these applications.

(a) Branch Mispredicts per 1K instructions (MPKI) and Cache Average Access Time (AAT) across applications.

(b) A-stream utilization for slipstream processors 2.0.

(c) Speedup of Slipstream 2.0 DBP+DLP relative to the baseline.

Figure 8.13 Slipstream 2.0 as microarchitectural turbo boost.

Hence using these simple heuristics DBP and DLP can benefit low-IPC applications effectively without incurring power cost for other high-IPC applications or applications that do not benefit from DBP and DLP.

8.6 IR-mispredictions

IR-misprediction is whenever an A-stream branch outcome was found to be wrong by the R-stream core. When this happens both cores are flushed and a re-sync is triggered. The cost of an IR-misprediction is much higher compared to a branch misprediction as both

the pipelines are squashed, register state is copied from one core to the other, and the pipeline is refilled. The penalty in number of cycles is 3x more for an IR-misprediction compared to a conventional branch misprediction. Hence it is necessary to make sure we don't IR-mispredict often. The total IR-mispredictions observed for the proposed schemes are shown in Figure 8.14. We observe an average IR-misprediction rate of less than one every 1000 instructions. Despite the removal of load dependent forward branches, we do not remove backward loop branches as it could impact forward progress of A-stream's execution. If there are load dependent backward branches in the A-stream, they could cause an IR-misprediction. But these occur mostly during loop exit conditions that occur when the trip count of the loop is completed. Hence it doesn't significantly impact the prefetching effect of the A-stream thread.

Figure 8.14 IR-Mispredictions per 1K instructions (MPKI) in Slipstream 2.0.

8.7 Delay Buffer Occupancy

Figure 8.15 shows the delay buffer occupancy for all the schemes and DCE. With DBP, as A-stream is slowed down by cache misses the delay buffer is empty for a good fraction of execution time. For misprediction heavy applications number of empty cycles is reduced

due to DBP pre-execution. This is seen from benchmarks *hmmmer*, *bzip2* and *astar*. As DCE suffers through mispredictions for these applications we see more empty delay buffer cycles for these benchmarks with DCE. DLP and DBP+DLP show much higher delay buffer occupancy. As A-stream converts load misses to prefetches, it is rarely slowed down and is able to provide branch outcomes readily to the R-stream. We can also see some fraction of cycles where the delay buffer is full for DLP and DBP+DLP. These are cases where A-stream is sped up and R-stream is the bottleneck. Although this scenario is not common as the R-stream usually keeps up as we can see that the delay buffer is neither empty nor full for most cycles showing both the streams are running in tandem.

Figure 8.15 Delay Buffer Occupancy in Slipstream 2.0.

CHAPTER

9

ENERGY

Figure 9.1 shows energy and energy-delay-product (EDP) of Slipstream 2.0 (two cores) normalized to the baseline (one core). Despite using two cores and additional (albeit quite small) components, the average energy expenditure is 4% less than the baseline with a single core. The main reason is that the reduced execution time leads to lower static energy. There is an increase in dynamic energy as a result of redundant execution. But the benefit from reduced static energy outweighs the increase in dynamic energy significantly.

Figure 9.1 Energy and EDP of Slipstream 2.0 (dual cores) normalized to baseline (single core).

In astar, bzip2, and hmmer, we expend 12% to 24% more energy, but if we factor-in the speedups and measure EDP we see that they do better than the baseline. Memory-bound applications that benefited the most from DLP see a significant drop in energy. Their EDP is much lower than the baseline as well. On average, Slipstream 2.0 reduces total energy by 4% and EDP by 43% compared to the baseline.

CHAPTER

10

CONCLUSION

This dissertation presented Slipstream 2.0, a new pre-execution microarchitecture that meets four criteria: (i) retains the simpler coordination of a leader-follower microarchitecture as compared to per-dynamic-instance helper threads, (ii) is fully automated with just hardware, (iii) targets both branches and loads, (iv) is effective in exploiting that which is targeted. We reviewed prior pre-execution proposals and showed none of them meet all four criteria. Slipstream 2.0’s key innovation in the space of leader-follower architectures is to remove the *forward control-flow slices* of pre-executable delinquent branches and delinquent loads, from the leading thread.

In addition to being the first pre-execution proposal that meets all four criteria, it includes a simple auto-enable/disable mechanism making it a useful microarchitectural turbo-boost feature and it economically reduces the leading thread by a simple adaptation

of conventional branching (“branch-to-reconvergent-point” instead of “branch-to-target”).

We compared Slipstream 2.0 to the baseline single core and the only other hardware-only leader-follower prior works in pre-execution: Slipstream (targets branches) and Dual Core Execution (DCE) (targets loads). For SPEC 2006/2017 SimPoints wherein Slipstream 2.0 is auto-enabled, it achieves geomean speedups of 67%, 60%, and 12%, over baseline, Slipstream, and DCE. For SimPoints with primarily delinquent branches, Slipstream 2.0 is 34%, 23%, and 21%, faster than baseline, Slipstream, and DCE. For SimPoints with primarily delinquent loads, Slipstream 2.0 is 84%, 73%, and 9%, faster than baseline, Slipstream, and DCE. It gives an average reduction of 43% in Energy Delay Product (EDP) and 4% in energy compared to baseline. Finally, it adds only 4.2KB in storage cost for the IR-detector, IR-predictor, and Delay Buffer.

BIBLIOGRAPHY

- [Akk03] Akkary, H. et al. "Checkpoint processing and recovery: towards scalable large instruction window processors". *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36*. 2003, pp. 423–434.
- [Bur97] Burger, D. et al. "Datascalar Architectures". *Conference Proceedings. The 24th Annual International Symposium on Computer Architecture*. 1997, pp. 338–349.
- [Cha99] Chappell, R. S. et al. "Simultaneous subordinate microthreading (SSMT)". *Proceedings of the 26th International Symposium on Computer Architecture (Cat. No.99CB36367)*. 1999, pp. 186–195.
- [Cha02] Chappell, R. S. et al. "Difficult-path branch prediction using subordinate microthreads". *Proceedings 29th Annual International Symposium on Computer Architecture*. 2002, pp. 307–317.
- [Che17] Chen, Q. et al. "Prophet: Precise QoS Prediction on Non-Preemptive Accelerators to Improve Utilization in Warehouse-Scale Computers". *SIGOPS Oper. Syst. Rev.* **51.2** (2017), pp. 17–32.
- [Col01] Collins, J. D. et al. "Speculative precomputation: long-range prefetching of delinquent loads". *Proceedings 28th Annual International Symposium on Computer Architecture*. 2001, pp. 14–25.
- [Col04] Collins, J. D. et al. "Control Flow Optimization Via Dynamic Reconvergence Prediction". *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 37. Portland, Oregon: IEEE Computer Society, 2004, pp. 129–140.
- [CH99] Connors, D. A. & Hwu, W.-m. W. "Compiler-directed Dynamic Computation Reuse: Rationale and Initial Results". *Proceedings of the 32Nd Annual ACM/IEEE International Symposium on Microarchitecture*. MICRO 32. Haifa, Israel: IEEE Computer Society, 1999, pp. 158–169.
- [Dub95] Dubey, P. K. et al. "Single-program Speculative Multithreading (SPSM) Architecture: Compiler-assisted Fine-grained Multithreading". *Proceedings of the IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques*. PACT '95. Limassol, Cyprus: IFIP Working Group on Algol, 1995, pp. 109–121.
- [For15a] Forbes, E. et al. "Under 100-cycle Thread Migration Latency in a Single-ISA Heterogeneous Multi-core Processor". *Poster session of Hot Chips*. 2015.

- [For15b] Forbes, E. et al. “Experiences with Two FabScalar-based Chips”. *6th Workshop on Architectural Research Prototyping (WARP-6)*. 2015.
- [GB05] Ganusov, I. & Burtcher, M. “Future execution: a hardware prefetching technique for chip multiprocessors”. *14th International Conference on Parallel Architectures and Compilation Techniques (PACT’05)*. 2005, pp. 350–360.
- [GH08] Garg, A. & Huang, M. C. “A performance-correctness explicitly-decoupled architecture”. *2008 41st IEEE/ACM International Symposium on Microarchitecture*. 2008, pp. 306–317.
- [Gar11] Garg, A. et al. “Speculative Parallelization in Decoupled Look-ahead”. *2011 International Conference on Parallel Architectures and Compilation Techniques*. 2011, pp. 413–423.
- [Has16] Hashemi, M. et al. “Continuous runahead: Transparent hardware acceleration for memory intensive workloads”. *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2016, pp. 1–12.
- [Jac96] Jacobsen, E. et al. “Assigning confidence to conditional branch predictions”. *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*. 1996, pp. 142–152.
- [Kan18] Kannan, R. S. et al. “Proctor: Detecting and Investigating Interference in Shared Datacenters”. *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2018, pp. 76–86.
- [Kan19a] Kannan, R. S. et al. “Caliper: Interference Estimator for Multi-tenant Environments Sharing Architectural Resources”. *ACM Trans. Archit. Code Optim.* **16.3** (2019), 22:1–22:25.
- [Kan19b] Kannan, R. S. et al. “GrandSLAM: Guaranteeing SLAs for Jobs in Microservices Execution Frameworks”. *Proceedings of the Fourteenth EuroSys Conference 2019*. EuroSys ’19. Dresden, Germany: ACM, 2019, 34:1–34:16.
- [Kir05] Kirman, N. et al. “Checkpointed early load retirement”. *11th International Symposium on High-Performance Computer Architecture*. 2005, pp. 16–27.
- [Kla98] Klauser, A. et al. “Dynamic hammock predication for non-predicated instruction set architectures”. *Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.98EX192)*. 1998, pp. 278–285.
- [KH18a] Kondguli, S. & Huang, M. “Bootstrapping: Using SMT Hardware to Improve Single-Thread Performance”. *IEEE Computer Architecture Letters* **17.2** (2018), pp. 205–208.

- [KH18b] Kondguli, S. & Huang, M. “R3-DLA (Reduce, Reuse, Recycle): A More Efficient Approach to Decoupled Look-Ahead Architectures”. *CoRR* **abs/1812.04514** (2018). arXiv: 1812.04514.
- [KR04] Koppanalil, J. J. & Rotenberg, E. “A Simple Mechanism for Detecting Ineffective Instructions in Slipstream Processors”. *IEEE Trans. Comput.* **53.4** (2004), pp. 399–413.
- [Kum03] Kumar, R. et al. “Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction”. *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*. 2003.
- [Kum04] Kumar, R. et al. “Single-ISA heterogeneous multi-core architectures for multi-threaded workload performance”. *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*. 2004.
- [Li09] Li, S. et al. “McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures”. *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2009, pp. 469–480.
- [MMR07] Mesa-Martinez, F. & Renau, J. “Effective Optimistic-Checker Tandem Core Design through Architectural Pruning”. *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. 2007, pp. 236–248.
- [Mos01] Moshovos, A. et al. “Slice-processors: An Implementation of Operation-based Prediction”. *Proceedings of the 15th International Conference on Supercomputing*. ICS ’01. Sorrento, Italy: ACM, 2001, pp. 321–334.
- [Mut03] Mutlu, O. et al. “Runahead execution: an alternative to very large instruction windows for out-of-order processors”. *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings*. 2003, pp. 129–140.
- [Nag12a] Nagarajan, V. et al. “Compilation Accelerator on Silicon”. *2012 IEEE Computer Society Annual Symposium on VLSI*. 2012, pp. 267–272.
- [Nag12b] Nagarajan, V. et al. “SCOC IP Cores for Custom Built Supercomputing Nodes”. *2012 IEEE Computer Society Annual Symposium on VLSI*. 2012, pp. 255–260.
- [Nag13] Nagarajan, V. et al. “Performance and energy efficient cache system design: Simultaneous execution of multiple applications on heterogeneous cores”. *2013 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 2013, pp. 200–205.
- [PH14] Parihar, R. & Huang, M. C. “Accelerating decoupled look-ahead via weak dependence removal: A metaheuristic approach”. *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 2014, pp. 662–677.

- [Pur00] Purser, Z. et al. "A study of slipstream processors". *Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-33 2000*. 2000, pp. 269–280.
- [Pur02] Purser, Z. et al. *Slipstream Memory Hierarchies*. Tech. rep. CESR-TR-02-3. 2002.
- [Red06] Reddy, V. K. et al. "Understanding Prediction-based Partial Redundant Threading for Low-overhead, High- Coverage Fault Tolerance". *SIGARCH Comput. Archit. News* **34.5** (2006), pp. 83–94.
- [Rot99] Rotenberg, E. "AR-SMT: a microarchitectural approach to fault tolerance in microprocessors". *Digest of Papers. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (Cat. No.99CB36352)*. 1999, pp. 84–91.
- [RS01] Roth, A. & Sohi, G. S. "Speculative data-driven multithreading". *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*. 2001, pp. 37–48.
- [Sez16] Seznec, A. "TAGE-SC-L Branch Predictors Again." *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*. 2016.
- [She12] Sheikh, R. et al. "Control-Flow Decoupling". *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. 2012, pp. 329–340.
- [She15] Shevgoor, M. et al. "Efficiently prefetching complex address patterns". *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2015, pp. 141–152.
- [Soh95] Sohi, G. S. et al. "Multiscalar processors". *Proceedings 22nd Annual International Symposium on Computer Architecture*. 1995, pp. 414–425.
- [Sri04] Srinivasan, S. T. et al. "Continual Flow Pipelines". *SIGOPS Oper. Syst. Rev.* **38.5** (2004), pp. 107–119.
- [Sri17] Srinivasan, V. et al. "H3 (Heterogeneity in 3D): A Logic-on-Logic 3D-Stacked Heterogeneous Multi-Core Processor". *2017 IEEE International Conference on Computer Design (ICCD)*. 2017, pp. 145–152.
- [Sri15] Srinivasan, V. *Phase II Implementation and Verification of the H3 Processor*. 2015.
- [Sri13] Srinivasan, V. et al. "Schmitt Trigger based SRAM Cell for Ultralow Power Operation- A CNFET based Approach". *Procedia Engineering* **64** (2013). International Conference on Design and Manufacturing (IConDM2013), pp. 115–124.

- [Sun00] Sundaramoorthy, K. et al. "Slipstream Processors: Improving Both Performance and Fault Tolerance". *SIGPLAN Not.* **35.11** (2000), pp. 257–268.
- [Wal91] Wall, D. W. "Limits of Instruction-level Parallelism". *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS IV. Santa Clara, California, USA: ACM, 1991, pp. 176–188.
- [Zho05] Zhou, H. "Dual-core execution: building a highly scalable single-thread instruction window". *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*. 2005, pp. 231–242.
- [ZS01] Zilles, C. & Sohi, G. "Execution-based prediction using speculative slices". *Proceedings 28th Annual International Symposium on Computer Architecture*. 2001, pp. 2–13.
- [ZS02] Zilles, C. & Sohi, G. "Master/Slave Speculative Parallelization". *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002. (MICRO-35). Proceedings*. 2002, pp. 85–96.
- [ZS00] Zilles, C. B. & Sohi, G. S. "Understanding the backward slices of performance degrading instructions". *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*. 2000, pp. 172–181.
- [Zil99] Zilles, C. B. et al. "The use of multithreading for exception handling". *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*. 1999, pp. 219–229.