ABSTRACT

KUMAR, CHANCHAL. Post-Silicon Microarchitecture. (Under the direction of Dr. Eric Rotenberg).

Improvement in single-thread performance of superscalar Out-of-Order microprocessors has slowed owing to technology limitations and power constraints. With stagnating frequency, continued performance improvement requires microarchitectural breakthroughs to unlock and harvest the IPC (instructions per cycle) potential of any given workload. However, microarchitectural features like Out-Of-Order execution and branch prediction, which have traditionally provided generational performance improvements across a wide range of benchmarks and application behaviors, have started showing diminishing returns. As a result, researchers have turned to strategies that pinpoint specific application behaviors, leading to exotic microarchitecture proposals which perform exceedingly well on some, but not all workload regions, while potentially even reducing performance for tasks adversely affected by the proposed microarchitecture additions. However, the narrow applicability of these specialized microarchitectural enhancements make it difficult to justify their inclusion in general-purpose processors.

We propose coupling reconfigurable logic (e.g., FPGA/CGRA) with a CPU core, on the same chip, via a simple non-intrusive interface to allow post-silicon deployment of microarchitectural ideas according to an application's individual needs. The interface allows communication between key stages of the pipeline and the reconfigurable logic, such that application-specific features (for example, custom branch predictors and prefetchers) targeting a specific region of interest in an application to unlock more ILP (Instruction-Level Parallelism) or MLP (Memory-Level Parallelism), can be designed in the reconfigurable logic and be able to influence the microarchitectural execution of the core in meaningful ways. This can be done, for example, by sending overriding custom branch predictions to the fetch unit of the core to get rid of branch misprediction bottlenecks, thus allowing the

core to reach its full IPC potential for the targeted workload phase. The CPU core still fetches and executes instructions in basically the same way while enhancements programmed into the reconfigurable hardware intervene when needed to unlock more performance in the instruction stream. The ability to instantiate microarchitecture components after fabrication, increases the value proposition of deploying microarchitecture ideas on an individual application basis. This leads to a novel microarchitecture paradigm, called Post-Silicon Microarchitecture (PSM), which fundamentally changes the trade-offs in implementing microarchitecture enhancements with narrow applicability.

We describe several examples of custom branch predictors and prefetchers targeted at bottlenecks found in many SPEC benchmarks, the issues with the round-trip latency of communication which crop up when developing microarchitectural components on the potentially slower reconfigurable fabric that needs to interact with the faster core, and potential mitigations which make this paradigm feasible.

© Copyright 2020 by Chanchal Kumar

All Rights Reserved

Post-Silicon Microarchitecture

by
Chanchal Kumar

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
2020

APPROVED BY:

Dr. Gregory T. Byrd

Dr. Huiyang Zhou

Dr. Michela Becchi

Dr. Eric Rotenberg
Chair of Advisory Committee

DEDICATION

Dedicated to Sameer & Saurav, who, either always, or never, believed in me.

I could never tell.

BIOGRAPHY

Chanchal Kumar was born in Bihar, India. He attended Saint Joseph's School, Bhagalpur, before joining the Electronics & Communication Engineering Department at Indian Institute of Technology, Roorkee. At IIT Roorkee, he worked on several projects related to digital design and computer hardware which sparked his interest in learning more about current high-performance computer systems, and helping build the next generation of microprocessors. After his undergraduate studies, Chanchal worked as a technical associate at a patent consultancy firm, before starting his doctoral studies at North Carolina State University under the supervision of Dr. Eric Rotenberg. During the course of his graduate studies, Chanchal completed two internships at Samsung Austin Research Center and worked on various research projects related to single-core and multi-core microprocessors. His first employment after PhD will be as a Senior Performance Architect at ARM, Austin.

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my sincerest gratitude to everyone who has extended their support and encouragement throughout my graduate studies. The list is huge; and while it may not be feasible for me to include everyone's name here, I hope that my indebtedness to their contributions, both to my academic as well as personal life, is apparent.

I am especially grateful to my advisor, Dr. Eric Rotenberg, whose emphasis on pursuing and solving critical problems of a fundamental nature has motivated me to be a better researcher. In the past, I have found myself leaning more towards derivative problems and being satisfied with incremental improvements, but his influence has resulted in a directional shift in how I now approach Computer Architecture research. My appreciation for his guidance in this regard cannot be emphasized enough. I am also thankful to him for letting me pursue my projects independently while providing much needed guidance when necessary.

I would also like to thank the members of my advisory committee for their constructive feedback and suggestions. I owe much of my knowledge to the amazing professors here at NC State and count myself fortunate to have had the opportunity to learn from them during my early graduate years. Being able to attend their classes has, by itself, made traveling half way across the world, away from all friends and family, worthwhile.

Thanks to Vinesh Srinivasan for his help with the simulator and evaluation frameworks. Acknowledgements are also due to the other members of the research group who have worked, or are working, on different aspects of the Post-Silicon Microarchitecture project - Shubham Bhawalkar, Aayush Chaudhary, Anirudh Seshadri, Utkarsh Mathur, Saransh Jain, and Adith Vastrad.

Sameer, Saurav, Mehrnaz, Sidharth, Bindhiya, Sahitya, Paritosh, Vamshi, Tharun, Nishant, and Suresh have collectively made these past 5 years, with the sometimes immense

pressure of graduate school, bearable, and almost fun. From the other side of the globe, Abhay, Abhishek, Aakash, Gautam, Nikhil, and Rohit have helped bridge the geographical gap and made me feel closer to home.

This dissertation would not have been possible without the support of my parents, my sister, and my brother. My passion to learn was ignited early at home and I am forever grateful for everything they have done for me. A special thanks also goes out to my brother-in-law for providing me a broader perspective on many things, whenever I needed it.

Finally, I would like to extend my acknowledgement to several people from the industry who have helped expand my knowledge about Computer Architecture beyond what one normally learns in academia - Tarun, Ned, Anoop, Tom, Ahmed, and Nithesh from when I interned at SARC, and Jayesh Gaur and Rangeen Basu Roy Chowdhury from the regular meetings regarding the Post-Silicon Microarchitecture project.

This research 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	xii
Chapter 1 INTRODUCTION	1
1.1 Motivation	1
1.2 Objective	2
1.3 Contributions	5
1.4 Outline	6
Chapter 2 PSM Architecture	8
2.1 High-Level Overview	8
2.2 PSM-Agent	10
2.2.1 Retire Snoop Table (RST)	11
2.2.2 Observation Queue at Retire (ObsQ-R)	13
2.2.3 Intervention Queue at Fetch (IntvQ-F)	13
2.2.4 Fetch Snoop Table (FST)	14
2.2.5 Intervention Queue at Issue (IntvQ-IS)	15
2.2.6 Observation Queue at Execute (ObsQ-EX)	17
2.2.7 Checkpoint	17
2.3 Baseline Core Modifications	17
Chapter 3 PSM Use Cases	23
3.1 Decoupled custom EXACT branch predictor	25
3.1.1 Analysis	28
3.1.2 PSM design	28
3.1.3 Evaluation Methodology	31
3.1.4 Results	32
3.2 Control Flow Decoupling	38
3.2.1 Analysis	39
3.2.2 PSM design	40
3.2.3 Results	42
3.3 Custom Branch Predictor for <i>wrf</i>	46
3.3.1 Analysis	48
3.3.2 PSM design	49
3.3.3 Results	49
3.4 Load Prefetching	51
3.4.1 <i>libquantum</i>	53
3.4.2 <i>lbm</i>	54
3.4.3 <i>bwaves</i>	55
3.4.4 <i>milc</i>	55

3.4.5	<i>leslie</i>	56
3.4.6	Results	56
3.5	Load-dependent Load Prefetching	58
3.5.1	Results	61
Chapter 4	Discussion	63
4.1	Multi-programming	64
4.2	Choice of Reconfigurable Fabric	65
4.3	Programming & source-code recompilation	65
4.4	Other use cases & updates to the PSM Interface	66
4.5	Automation	68
Chapter 5	Related Work	70
5.1	Reconfigurable fabric for acceleration	70
5.1.1	Tightly-integrated reconfigurable functional units	71
5.1.2	Reconfigurable ‘co-processors’	72
5.1.3	Reconfigurable fabric in CMPs	72
5.2	Prior work related to PSM Use Cases	73
Chapter 6	Conclusion	75
References		77

LIST OF TABLES

Table 2.1	Baseline Core Configuration	18
Table 3.1	Misprediction rate and MPKI for the delinquent branches in <i>astar</i> . . .	28
Table 3.2	Key for different PSM parameters used in the sensitivity studies	32
Table 3.3	Misprediction rate for delinquent branches in <i>astar</i> for a PSM design (Configuration: clk4_w4, delay4, queue32, portALL with 32KB predictor tables, 512-entry worklists, and 32-entry undo log).	33
Table 3.4	Misprediction rate for delinquent branches in <i>astar</i> for different predictor sizes.	35
Table 3.5	Misprediction rate for delinquent branches in <i>astar</i> for different worklist sizes.	36
Table 3.6	Misprediction rate for CFD-targeted branches in <i>astar</i>	42
Table 3.7	Misprediction rate for delinquent branches in the CFD use case for different index table sizes.	47
Table 3.8	Misprediction rate and MPKI of the delinquent branches in <i>wrf</i>	49

LIST OF FIGURES

Figure 1.1	Performance potential for some SPEC benchmarks with perfect branch prediction or perfect data cache, normalized against a baseline with realistic branch predictors and data cache.	3
Figure 2.1	High-level overview of the PSM architecture.	9
Figure 2.2	Actions taken by the core's fetch unit for a matching entry in FST. . .	16
Figure 2.3	Custom branch prediction by PSM-Agent overrides default prediction by core.	19
Figure 2.4	Instructions are fetched from the PSM-Agent when the core is in the 'Enable Instruction Fetch' mode.	20
Figure 2.5	Opportunistic execution of load or prefetch OPs sent by PSM-RE. . .	21
Figure 3.1	Custom decoupled branch predictor design for <i>astar</i>	30
Figure 3.2	Performance of custom branch predictor for <i>astar</i> , for different <i>C</i> and <i>W</i> parameters. All configs are delay0, queue32, portALL, and have 32KB predictors, 512-entry worklists, and a 32-entry undo log. . . .	34
Figure 3.3	Performance of custom branch predictor for <i>astar</i> , for different <i>D</i> and <i>Q</i> parameters. (a) All configs are clk4_w4 and queue32. (b) All configs are clk4_w4 and delay4. All configs in (a) and (b) are portALL, and have 32KB predictors, 512-entry worklists, and a 32-entry undo log.	34
Figure 3.4	Performance of custom branch predictor for <i>astar</i> , for different values of the <i>P</i> parameter and different number of entries in the undo log. (a) All configs are clk4_w4, delay4, queue32, with a 32-entry undo log. (b) All configs are clk4_w4, delay4, queue32, and portALL. All configs in (a) and (b) have 32KB predictors and 512-entry worklists.	35
Figure 3.5	(a) Performance of custom branch predictor for <i>astar</i> , for different sizes of the predictor tables (all configs are clk4_w4, delay4, queue32, portALL, and have 512-entry worklists and a 32-entry undo log). (b) Performance of custom branch predictor for <i>astar</i> , for different number of entries in the worklist tables (all configs are clk4_w4, delay4, queue32, portALL, and have 32KB predictors).	36
Figure 3.6	Performance of a dedicated Oracle predictor for <i>astar</i> (configured for similar accuracy as the PSM design with 12KB predictors), for different values of the 'restart delay'.	37
Figure 3.7	Performance of Control Flow Decoupling, for different <i>C</i> and <i>W</i> parameters. All configs are delay0, queue32, chkpt32, portALL, with an 8KB index table. Legend available in Section 3.1.3.	43
Figure 3.8	Performance of CFD, for different <i>D</i> and <i>Q</i> parameters. (a) All configs are clk4_w4, queue32, chkpt32, portALL. (b) All configs are clk4_w4, delay4, chkpt32, portALL. All configs in (a) and (b) use an 8KB index table. Legend available in Section 3.1.3.	43

Figure 3.9	Performance of CFD, for different N and P parameters. (a) All configs are clk4_w4, delay4, queue32, portALL. (b) All configs are clk4_w4, delay4, queue32, chkpt32. All configs in (a) and (b) use an 8KB index table. Legend available in Section 3.1.3.	44
Figure 3.10	Performance of Control Flow Decoupling (with loop buffer), for different C and W parameters. All configs are delay0, queue32, chkpt32, portALL, with an 8KB index table. Legend available in Section 3.1.3.	45
Figure 3.11	Performance of CFD (with loop buffer), for different D and Q parameters. (a) All configs are clk4_w4, queue32, chkpt32, portALL. (b) All configs are clk4_w4, delay4, chkpt32, portALL. All configs in (a) and (b) use an 8KB index table. Legend available in Section 3.1.3.	46
Figure 3.12	Performance of CFD (with loop buffer), for different N and P parameters. (a) All configs are clk4_w4, delay4, queue32, portALL. (b) All configs are clk4_w4, delay4, queue32, chkpt32. All configs in (a) and (b) use an 8KB index table. Legend available in Section 3.1.3.	46
Figure 3.13	Performance of CFD (with loop buffer) for different sizes of the index table. All configs are clk4_w4, delay4, queue32, chkpt32, portALL. Legend available in Section 3.1.3.	47
Figure 3.14	Performance of the PSM design for <i>wrf</i> , for different C and W parameters. All configs are delay0, queue32, portALL. Legend available in Section 3.1.3.	50
Figure 3.15	Performance of the PSM design for <i>wrf</i> , for different D parameters. All configs are clk4_w4, queue32, portALL. Legend available in Section 3.1.3.	50
Figure 3.16	Performance of the PSM design for <i>wrf</i> , for different Q and P parameters. (a) All configs are clk4_w4, delay4, portALL. (b) All configs are clk4_w4, delay4, queue32. Legend available in Section 3.1.3.	51
Figure 3.17	PSM design for load prefetching.	52
Figure 3.18	Uneven latency reduction for the delinquent loads in <i>lbm</i> , by the VLDP prefetcher.	55
Figure 3.19	Performance of PSM for load prefetching, for different C and W parameters. All configs are delay0, queue32, portALL. Legend available in Section 3.1.3.	57
Figure 3.20	Performance of PSM for load prefetching, for different D parameters. All configs are clk4_w4, queue32, portALL. Legend available in Section 3.1.3.	58
Figure 3.21	Performance of PSM for load prefetching, for different Q parameters. All configs are clk4_w4, delay4, portALL. Legend available in Section 3.1.3.	58
Figure 3.22	Performance of PSM for load prefetching, for different P parameters. All configs are clk4_w4, delay4, queue32. Legend available in Section 3.1.3.	59

Figure 3.23	Performance of PSM for <i>libquantum</i> , with an adaptive prefetch distance (starting from an initial distance of 60), compared to static prefetch distances. Configuration is clk4_w4, delay0, queue32, portALL. Legend available in Section 3.1.3.	59
Figure 3.24	Load-dependent load prefetching for <i>mcf</i>	61
Figure 3.25	Performance of the PSM design for <i>mcf</i> , with different <i>C</i> and <i>W</i> parameters. All configs are delay0, queue32, portALL. Legend available in Section 3.1.3.	62
Figure 3.26	Performance of the PSM design for <i>mcf</i> , with different <i>D</i> , <i>Q</i> , and <i>P</i> parameters. (a) All configs are delay0, queue32, portALL. (b) All configs are clk4_w4, delay4, portALL. (c) All configs are clk4_w4, delay4, queue32. Legend available in Section 3.1.3.	62

LIST OF LISTINGS

Listing 3.1	<i>astar's</i> ROI in which PSM is enabled.	26
Listing 3.2	The <i>makebound2</i> function with the load-dependent branches.	26
Listing 3.3	<i>astar's</i> ROI targeted for Control Flow Decoupling.	40
Listing 3.4	Branch slice injected into the core for Control Flow Decoupling.	41
Listing 3.5	Region of interest in the <i>wrf</i> benchmark.	47
Listing 3.6	<i>libquantum's</i> quantum_toffoli function.	53
Listing 3.7	<i>mcf's</i> ROI with the delinquent loads.	60

CHAPTER

1

INTRODUCTION

1.1 Motivation

Microprocessors have historically seen unprecedented levels of year-over-year performance improvement owing to shrinking transistors and innovations in microarchitectures. Every new generation of microprocessor would bring increased frequency and efficiency, with new microarchitecture components and bigger caches that helped unlock more performance in application execution and bridge the memory gap. Microarchitects introduced out-of-order (OOO) instruction execution to target inherent instruction-level-parallelism (ILP) in all workloads, non-blocking caches to allow more memory-level-parallelism (MLP), and superscalar processor designs to be able to execute and retire multiple instructions in the same cycle. Addition of techniques like Branch Prediction allowed the out-of-order cores

to harvest ILP from a bigger instruction window and get more performance. Proactively prefetching data for an application, before it is requested, coupled with adding multiple levels in the cache hierarchy, helped reduce the execution time even further by allowing faster access to data when needed by an application. These improvements in the core microarchitecture were targeted at a wide range of benchmarks and provided good general performance across many diverse workloads.

However, improvement in single-thread performance of Out-of-Order (OOO) microprocessors has slowed owing to technology limitations and power constraints. On one hand, with the end of Dennard scaling, core frequency has stagnated. On the other hand, microarchitectural features like OOO execution and branch prediction, which have traditionally provided generational performance improvements across a wide range of benchmarks and application behaviors, have started showing diminishing returns. Increasing the superscalar width doesn't help because the processor is unable to expose and exploit additional ILP with the plateauing of branch prediction accuracy. At the same time, even aggressive prefetchers targeting complex address patterns are unable to hide the memory access latency completely. Figure 1.1 shows the performance gap for several SPEC benchmarks, between the high-performance Out-Of-Order superscalar core, used in our studies, with realistic branch predictors and prefetchers, and the same core augmented with a perfect branch predictor (which always predicts a branch's direction correctly) or a perfect data cache (where the load instructions always hit in the Level-1 data cache). The figure shows the performance potential of the core that remains unexploited because of the inefficiencies in current branch predictors and prefetchers.

1.2 Objective

As a result of the diminishing returns seen from traditional microarchitecture components, that, in general, try to target all workload execution phases, computer architecture research

Figure 1.1: Performance potential for some SPEC benchmarks with perfect branch prediction or perfect data cache, normalized against a baseline with realistic branch predictors and data cache.

has recently seen a shift towards proposals which try to identify and target specific kinds of workload phases by including microarchitectural features which spring into action when the targeted application behavior is detected, and, hopefully, stay dormant when not needed. This has led to some exotic microarchitecture proposals which work exceedingly well for some, but not all, workload regions, and occasionally even reduce performance for untargeted tasks adversely affected by the proposed microarchitectural feature. These specialized microarchitectural enhancements haven't seen eager adoption in commercial processor designs. Due to their narrow applicability, it is difficult to justify their inclusion in *general-purpose* processors. They cannot all be included, while at the same time, one or few of these proposed microarchitecture components cannot be included in CPU cores, which are expected to give good general performance across a multitude of workloads, because of their limited and specialized nature. An ideal solution to this dilemma of *whether* or not

to include the microarchitectural features that have narrow applicability, as well as *which* features to include, if need be, would be to *virtualize* some part of the hardware which can *morph* into any specialized microarchitectural component, as and when needed by any specific application.

Reconfigurable hardware, such as FPGA (Field Programmable Gate Array) [KTR08] or CGRA (Coarse-Grained Reconfigurable Architecture) [TK12] provide the capability to program, configure, and synthesize new designs in the same hardware, ‘on-the-fly’. If reconfigurable logic is integrated with a general-purpose core, it should be possible to develop customized microarchitecture components for an application, and synthesize the custom hardware into the reconfigurable logic when that application runs on the core. The synthesized design could intervene in the microarchitectural execution of the baseline core to help narrow the performance gap shown in Figure 1.1. For example, for the *astar* benchmark, suffering from branch misprediction bottlenecks, a custom branch predictor could be synthesized to get close to the performance of perfect branch prediction. Similarly, *lbm* might benefit from a heavily customized prefetcher to achieve performance close to having a perfect data cache.

This objective runs into several hurdles when considering a realistic design:

- A simple and non-intrusive, but efficient and powerful, interface is required between the core and the reconfigurable fabric. We want non-intrusiveness so as to not adversely affect the execution of the baseline core. While a custom design synthesized in the reconfigurable fabric might benefit by having access to many microarchitectural details of the core, the hardware complexity in such cases might become prohibitive. At the same time, the interface needs to be powerful enough to be able to influence the microarchitectural execution of the core in meaningful ways.
- Reconfigurable logic tends to run at a slower clock frequency compared to a baseline high-performance core. This makes implementing custom microarchitecture components, that interface with the core, difficult due to a potentially large round-trip

latency of communication between the faster core and the slower reconfigurable logic. For example, requesting and receiving custom branch predictions in the core from the design synthesized in the reconfigurable fabric can incur tens or hundreds of cycles of delay. The instruction execution can be severely impacted if the core has to wait for multiple cycles every time it needs to use a custom branch prediction, or communicate with the reconfigurable fabric. This introduces a requirement of *timeliness* in the custom microarchitecture components synthesized in the reconfigurable fabric.

1.3 Contributions

We propose coupling reconfigurable logic (e.g., FPGA/CGRA) with a CPU core, on the same chip, via a simple non-intrusive interface to allow *post-silicon* deployment of microarchitectural ideas according to an application’s individual needs. We develop an interface that allows the design synthesized in the reconfigurable fabric to *observe* and *modify* the microarchitectural state of the core as well as the cache hierarchy. The interface allows communication between key stages of the pipeline and the reconfigurable logic, such that application-specific features (for example, custom branch predictors and prefetchers) targeting a specific region of interest (ROI) in an application can be designed in the reconfigurable logic and be able to influence the *microarchitectural* execution of the core in meaningful ways by unlocking more instruction-level or memory-level parallelism. The CPU core still fetches and executes instructions in basically the same way while enhancements programmed into the reconfigurable hardware intervene when needed to unlock more performance in the instruction stream, thus allowing the core to reach its full IPC potential for the targeted region of interest.

We analyze several workloads from the SPEC benchmark suite [Hen06] [BLvK18] and identify the regions which suffer from microarchitectural bottlenecks such as a high branch

misprediction rate, or a high cache miss rate. We develop several customized designs to ease the bottleneck for these identified regions, for example, by sending overriding custom branch predictions to the fetch unit of the core to get rid of branch misprediction bottlenecks. We show how the requirements of timeliness can be achieved for the different types of use cases via development of *decoupled* designs that run ahead of the core’s instruction stream. With the help of decoupled custom microarchitecture components that can provide timely and very accurate predictions to the core, we can achieve good performance improvement over the baseline superscalar Out-Of-Order core.

The ability to instantiate microarchitecture components after fabrication, increases the value proposition of deploying microarchitecture ideas on an individual application basis. This leads to a novel microarchitecture paradigm, called Post-Silicon Microarchitecture (PSM), which fundamentally changes the trade-offs in implementing microarchitecture enhancements with narrow applicability.

1.4 Outline

In Chapter 2, we discuss the PSM architecture. We first provide a high-level overview of the overall PSM design and how it works in conjunction with the core. We then present a detailed description of the PSM interface, including the interface components and the mechanism of communication and microarchitectural intervention. We discuss the design and objective of the PSM interface along with the support needed from the baseline core.

Chapter 3 presents several use cases for the PSM design. We describe examples of custom branch predictors and custom prefetchers for several SPEC benchmarks, the issues which crop up when developing microarchitectural components on the potentially slower reconfigurable fabric that needs to interact with the faster core, and potential mitigations which make this paradigm feasible. We present results along with each use case to show the performance potential of PSM.

In Chapter 4 we discuss various high-level issues such as multi-programming and the choice of reconfigurable logic, while the related work is presented in Chapter 5. We conclude in Chapter 6.

CHAPTER

2

PSM ARCHITECTURE

2.1 High-Level Overview

Figure 2.1 shows the high-level diagram of PSM, where a reconfigurable fabric (PSM-RF) has been coupled with the OOO superscalar core. The OOO core provides good general-purpose performance but has untapped IPC potential for several applications which suffer from microarchitectural constraints, such as a high branch misprediction or cache miss rate. In these cases, a custom component can be synthesized in the flexible PSM-RF to remove the targeted microarchitectural bottleneck. However, while PSM-RF can offer a high degree of reconfigurability (depending on the choice of reconfigurable logic - FPGA, CGRA, or Custom), this flexibility would normally come at the expense of the clock speed that PSM-RF can support. Interfacing the potentially slower (but flexible) PSM-RF with the

Figure 2.1: High-level overview of the PSM architecture.

faster (but inflexible) core becomes difficult. To address this issue, an interface unit, called PSM-Agent (PSM-A), is added between the core and PSM-RF.

PSM-A is tightly coupled with the core, has limited configurability (thus can run at the core's clock frequency), and acts as a communication medium between the core and PSM-RF. Designs synthesized in the PSM-RF need two capabilities, enumerated below, to influence the execution of an application.

1. PSM-RF should be able to *snoop* or *observe* key microarchitectural state from the core, and,
2. PSM-RF should be able to *intervene* and *modify* the microarchitectural state of the core in such ways that help remove the targeted microarchitectural constraints and improve performance.

PSM-A provides these capabilities via *Observation* queues (from core to PSM-RF) and *Intervention* queues (from PSM-RF to core) by allowing generic message-passing style of communication. There are two pairs of Observation and Intervention queues: one to

observe and modify the state of the core, and another to observe and modify the state of the caches. Push/pop of data into/from the queues can happen at potentially different clock frequencies depending on the frequency difference between the core and PSM-RF. For example, a custom branch predictor synthesized in PSM-RF can generate and send branch predictions to PSM-A using an Intervention Queue. PSM-A can then direct the core to override its own default prediction with the one provided by PSM-RF.

A configuration bitstream shipped with the executable synthesizes the custom microarchitecture in PSM-RF and configures the communication behavior of PSM-A. The next section describes the components of the PSM-Agent while also highlighting the support needed from the core.

2.2 PSM-Agent

The PSM-Agent is designed to be simple and non-intrusive to the core. While a custom design synthesized in PSM-RF might benefit by having access to many microarchitectural details of the core, the hardware complexity in such cases might become prohibitive. For example, a custom branch predictor in PSM-RF would benefit by being able to snoop the global branch history kept by the core, but the current interface limits the observation ports to a few key pipeline stages to reduce intrusiveness and retain simplicity. PSM-A is also designed to be agnostic to the choice of the reconfigurable logic used in PSM-RF.

PSM-A has several components which are described below. The Observation and Intervention queues are used for communicating data to and from PSM-RF, while the *Snoop Tables* act as controllers which dictate how the queue payloads are constructed, as well as how the core's microarchitectural behavior is changed. Changes required in the baseline core to support the PSM interface is summarized in Section 2.3.

2.2.1 Retire Snoop Table (RST)

The Retire Snoop Table (RST) is placed near the Retire stage of the core and is configured to allow PSM-RF to snoop key information from the retiring instructions. Each RST entry is composed of the following fields:

- **PC:** The PCs are of instructions that PSM-RF is interested in snooping. These are obtained via profiling of the application and assembly. When an instruction retires, its PC is checked against the entries in RST. If a matching entry is found, PSM-A constructs a payload and pushes it in the Retire Observation Queue (described in Section 2.2.2).
- **Three payload-type bits:** These include the 'Branch', 'Destination Register', and 'Store' payload-type bits. These bits determine what information is included in the constructed payload. If the 'Branch' bit is set in the matching entry, then the payload for the retiring (branch) instruction includes the actual T/NT direction of the branch. Similarly, for the 'Destination Register' bit, the payload includes the value of the destination register of the retiring instruction. If the 'Store' bit is set, PSM-A snoops the store value of the retiring (store) instruction, from the head of the Store Queue, and adds it into the payload.
- **Six configuration bits:** The configuration bits of the matching entry are responsible for changing the mode of execution of the baseline core, as well as configuring the communication behavior of PSM-A. The configuration bits are explained below:
 1. **Enable PSM mode:** This enables the communication queues and signifies the start of the region of interest (ROI) targeted by the custom design in PSM-RF.
 2. **Custom BP:** If the matching entry has this configuration flag enabled, then the core is directed to start consulting PSM-A to get custom branch predictions generated by PSM-RF (overrides core's default branch prediction).

3. ***Full Squash mode***: This squashes the pipeline and directs the core to do full squashes (from head of ROB) for branch mispredictions, instead of partial squashes (from middle of ROB), until PSM is disabled. When in this mode, PSM-A sends a packet to PSM-RF for all squashes and stalls fetch until an Acknowledgement is received in the Intervention Queue at Fetch (described in Section 2.2.3). This greatly simplifies the synchronization of the core and PSM-RF on a branch misprediction when PSM-RF is being used to generate custom branch predictions.
4. ***Disable PSM mode***: This signifies the end of the region of interest (ROI). It disables any features/mode enabled by PSM and returns the core to normal baseline execution.
5. ***Enable Instruction Fetch***: This squashes the core, takes a checkpoint of the architectural state (Section 2.2.7), and directs the core to start fetching a stream of instructions from PSM-A, instead of from the I-Cache. The ability to generate instruction slices in PSM-RF and have them execute in the core provides a very general method of changing the core's microarchitectural state. While in this mode, the store instructions (if any, sent by PSM-RF) are not exposed to the cache hierarchy and are simply discarded at retirement.
6. ***Disable Instruction Fetch***: This restores the architectural checkpoint and redirects the core to fetch instructions from the I-Cache. Because of the restoration of checkpoint which was taken when entering the 'Enable Instruction Fetch' mode, any instructions sent to the core by PSM-RF don't change the architectural behavior of the executing application.

The configuration bits are not mutually exclusive and can be combined together. For example, when PSM-RF is used to generate custom branch predictions, the entry for the start of the ROI can have the 'Enable PSM mode', 'Full Squash Mode', and 'Custom

BP' flags set. When the corresponding instruction retires, PSM is enabled, the core is squashed, and the fetch unit starts getting custom branch predictions from PSM-A, until the PSM mode is disabled at the end of ROI.

2.2.2 Observation Queue at Retire (ObsQ-R)

For each retiring instruction which has a matching entry in the RST, PSM-A constructs a payload and pushes it in this Observation Queue. When in the 'Full Squash' mode, a packet is sent to PSM-RF for each pipeline squash as well, allowing the synthesized design to take any necessary recovery actions. The payload consists of the PC, the configuration bits, a T/NT flag (if 'Branch' bit was set), a value field (if 'Destination Register' or 'Store' bit was set), and a 'squash' flag. PSM-RF pops payloads from this queue and uses them according to the synthesized hardware.

2.2.3 Intervention Queue at Fetch (IntvQ-F)

PSM-RF uses this intervention queue to send commands to the fetch unit of the core/PSM-A. The payload includes PC, a command, and the corresponding data. The different possible commands, and the corresponding action taken by the core, are briefly described below:

- **BRANCH_DIR:** This command indicates that the payload includes the custom branch prediction (T/NT direction) generated by the design synthesized in PSM-RF, and that this custom prediction should be used to override the default branch prediction generated by the core's fetch unit.
- **SYNC:** This command is used as a synchronization point, for example, in a stream of custom branch predictions generated by PSM-RF. In cases of prediction underflow from PSM-RF, this command can be sent to indicate to the core's fetch unit that it should use the default branch predictions until a fetched instruction's PC matches this payload's PC.

- **DONE:** This indicates that PSM-RF has generated all predictions that it can and the core should use default predictions for further branches.
- **INSTRUCTION:** This command indicates that the payload includes an instruction generated by PSM-RF. In this case, the core gets the instruction from PSM-A instead of from the I-Cache (this command is sent while the core is in the 'Enable Instruction Fetch' mode).
- **ACK:** This is the acknowledgement sent by PSM-RF after receiving the squash message from the core, signifying that corrective action has been completed by the design synthesized in PSM-RF.

2.2.4 Fetch Snoop Table (FST)

Each FST entry stores the PC of a branch instruction that is targeted by PSM for custom branch prediction, or the PC of the instruction used as the synchronization point. When 'Custom BP' mode is enabled, PSM-A searches the FST and head of IntvQ-F with the PC of the fetched instructions. Different actions can be taken for an instruction depending on whether or not a matching entry was found. These actions are shown in Figure 2.2 and described below:

- If the fetched instruction doesn't match with any FST entry, this means that it was not targeted for any intervention by the design synthesized in PSM-RF. In this case, the instruction continues its normal baseline execution.
- If a matching entry is found in FST, and the entry at the head of IntvQ-F also matches the fetched instruction, the custom branch prediction from the payload at head of IntvQ-F is used to override the default branch prediction (if command in payload at head was `BRANCH_DIR`). PSM-A also pops the entry at head of IntvQ-F.

- If a matching entry is found in FST, but the entry at head of IntvQ-F doesn't match, it means that the core's fetch stream has diverted from what PSM-RF expects and the core simply uses the default prediction, without popping the entry at the head of the queue. This diversion can be corrected by PSM-RF at the next pipeline squash, if necessary.
- If a matching entry is found in FST, but IntvQ-F is empty, it suggests that the custom branch prediction stream from PSM-RF has been delayed. In this case, fetch is stalled to wait for the custom prediction rather than using the default prediction. This helps keep the PSM-RF's expectation of program execution in sync with the core, while avoiding an expensive pipeline squash in case the default prediction is incorrect. This suggests that the PSM design should only target high MPKI (mispredictions per kilo instructions) branches for which the default branch predictor performs poorly. This also suggests that any custom branch predictor implemented in PSM-RF needs to be timely in providing its generated predictions to the core so as to not stall the core front-end.
- If the command at the head of IntvQ-F is SYNC, the rules itemized above still apply. This is used simply to skip providing custom predictions until a synchronization point is reached in the core's instruction stream. This might be necessary if the custom predictions from PSM-RF underflow, for example, due to insufficient resources. The SYNC entry is popped from the head of IntvQ-F when the corresponding synchronization instruction is fetched.

2.2.5 Intervention Queue at Issue (IntvQ-IS)

This intervention queue, along with the Observation Queue at Execute (Section 2.2.6), is used by PSM-RF to observe and modify the microarchitectural state of the caches. Components synthesized in the PSM-RF can send prefetch or load OPs to the core, via this

Figure 2.2: Actions taken by the core's fetch unit for a matching entry in FST.

intervention queue, for opportunistic execution. Each payload contains: 1) a command (LOAD or PREFETCH), 2) the address, and 3) the data size (for LOAD only). On a bubble in the load execution lane, the core picks and issues the OP at the head of IntvQ-IS. The issued OP flows through the normal load pipe (address translation and data cache lookup), but stays pinned at the head of IntvQ-IS until the OP resolves. The resolution of an OP is determined as follows:

- For prefetches, the OP is considered resolved if the cache line is already available in the data cache, or an MSHR entry is allocated if the lookup missed. The OP gets replayed from the head of IntvQ-IS on the next bubble, if it missed in the cache but an MSHR could not be allocated.
- For loads, the OP is considered resolved on a cache hit. In case of a miss, it gets replayed from the head of IntvQ-IS on the next bubble.

The entry at the head of IntvQ-IS is popped when the OP resolves. While this design means that at any given time, only one OP from IntvQ-IS can be issued/executed, it helps keep the hardware simple.

2.2.6 Observation Queue at Execute (ObsQ-EX)

If the core executed a load OP received from PSM-A, it pushes the loaded value into this observation queue, to be used by the design synthesized in PSM-RE. This allows observation of the cache's microarchitectural state by the PSM-RE. The payload consists of only 1 field, the loaded value.

2.2.7 Checkpoint

The PSM-Agent provides a mechanism to take a checkpoint of the core's architectural state when the 'Enable Instruction Fetch' mode (Section 2.2.1) is enabled. For checkpointing the register values, after the core is squashed, PSM-A uses the PRF ports that it is allowed to arbitrate on (described in Section 2.3) to copy over the contents of the architectural registers in multiple cycles. This checkpoint is restored into the core when the 'Instruction Fetch' mode is disabled. The application running on the core doesn't make any architectural progress while the instructions are fetched from PSM-A, but the microarchitectural state can be significantly modified to help improve performance.

2.3 Baseline Core Modifications

The baseline core is a 10-stage (Fetch-to-Retire) superscalar OOO core. The configuration is shown in Table 2.1. There are a few modifications needed in the baseline to properly interface with the PSM-Agent. These modifications need to be minimal and non-intrusive to prevent any negative impacts of tightly coupling the PSM-Agent with the core. We briefly described all baseline changes below:

Table 2.1: Baseline Core Configuration

Core	10-stage (Fetch to Retire) Out-of-Order; 4 instr./cycle Fetch/Retire; 8 instr./cycle Issue/Execute; ALUs: 4 Simple, 2 LS, 2 FP-Complex; ROB/IQ/LDQ/STQ/PRF: 224/100/72/72/288 entries
Branch Predictor	BP: 64KB TAGE-SC-L predictor [Sez16]; BTB: 4K entries, 4-way set-associative; RAS: 32 entries
Memory Hierarchy	L1: split, 32KB each, 8-way set-associative, 4-cycle access latency, next-N-line (N=2) prefetcher; L2: unified, 256KB, 8-way set-associative, 12-cycle access latency; L3: unified, 8 MB, 16-way set-associative, 42-cycle access latency; L2/L3 prefetcher: VLDP (5.5 Kb) [SKB ⁺ 15]; DRAM: 250-cycle access latency

- **Overriding branch prediction:** When a branch instruction is fetched, the core must consult PSM-A to see if it has a custom branch prediction available which overrides the default prediction by the core (Section 2.2.1). As shown in Figure 2.3, the PSM-Agent is looked up in parallel with the default branch predictor. If a custom prediction for the conditional branch is available, it overrides the default prediction. The rest of the branch prediction pipe remains the same. The current interface doesn't provide support for custom indirect branch predictions from the PSM-Agent, but it can be added in a future iteration of the PSM interface with minimal modifications.
- **Stall fetch:** The core should be able to stall fetch at the direction of PSM-A. This might happen, for example, if PSM-A expects a custom branch prediction from PSM-RF, for the fetched branch instruction, but it is delayed/unavailable (discussed in Section 2.2.4).
- **Fetch instructions from PSM-A:** The core allows a configuration ('Enable Instruction

Figure 2.3: Custom branch prediction by PSM-Agent overrides default prediction by core.

Fetch' in Section 2.2.1) where it fetches a stream of instructions sent by the design synthesized in PSM-RF via the Intervention Queue at Fetch (Section 2.2.3), instead of fetching instructions from the Instruction Cache. This is shown in Figure 2.4.

- **Full Squash mode:** Recovery on a branch misprediction, if any, in the baseline core is normally initiated as soon as the branch instruction executes. If the mispredicting branch instruction is in the middle of the ROB, the core does a partial squash where only the instructions after the mispredicting branch are flushed. When interfacing with a PSM-Agent, the core provides an alternate configuration where the squash is deferred until the branch reaches the ROB head and is ready to retire. This configuration can be enabled by PSM-A within an ROI, as needed (Section 2.2.1). This is done in order to simplify synchronizing with PSM-RF on misprediction events. Without the 'Full Squash' mode, the PSM-RF would need very intrusive hooks into the core microarchitecture to fix its predictor states on partial squashes. The use of full squashes on a misprediction, when custom branch predictions supplied by PSM-RF

Figure 2.4: Instructions are fetched from the PSM-Agent when the core is in the ‘Enable Instruction Fetch’ mode.

are being used by the core, suggests a need to have a high branch prediction accuracy to keep misprediction events rare and avoid the high penalty of a full pipeline squash.

- **Squash by PSM-A:** The core might be directed by PSM-A to squash the pipeline. This is generally done on mode switches, for example, when enabling or disabling PSM at ROI boundaries for custom branch prediction, or when enabling or disabling ‘Instruction Fetch’ mode’ (Section 2.2.1).
- **Execute Load/Prefetch OPs from PSM-A:** As mentioned in Section 2.2.5, the core needs to opportunistically pick and issue an OP from the head of IntvQ-IS on a bubble in the load execution lane. The OP sent by PSM-RF includes the load or prefetch address which is multiplexed directly in the load execution pipe after the address generation stage. This is shown in Figure 2.5. The OP follows the normal execution path of address translation and data cache lookup, so the design synthesized in PSM-RF can work with virtual rather than physical addresses. If the OP sent by PSM-RF was for a load, the data returned by the cache is pushed into ObsQ-EX (Section 2.2.6).

The core checks if ObsQ-EX has space available, before picking and issuing a load OP from IntvQ-IS.

Figure 2.5: Opportunistic execution of load or prefetch OPs sent by PSM-RF.

- PRF port sharing:** While preparing payloads to send to PSM-RF, PSM-A might need to read destination register values of retiring instructions from the PRF. This is done if the corresponding payload-type bit is set for the retiring instruction's RST entry (Section 2.2.1). We can avoid adding dedicated read ports for PSM-A in the PRF by doing opportunistic sharing of the already available read ports (with a lower priority for PSM-A than the baseline core). If any port is unused in a given cycle, PSM-A can opportunistically use it to read the register value out of the PRF. However, since doing a full arbitration across all PRF ports to find free/unused ports might suffer from timing constraints, we can statically, at design time, limit the arbitration to a small number of predetermined ports of the PRF. In our simulation model, we have a switch to select which execution lane's ports the PSM-Agent can arbitrate on. In Chapter 3, we present results for several use cases of PSM which show that opportunistically arbitrating on just one or two PRF ports allows us to find ample bubbles in PRF port usage, and obtain similar performance as when the arbitration is done across all ports.

- **Stall Retire:** The PSM-Agent can direct the core to stall the Retire stage in the following scenarios:
 - If PSM-A needs to send a payload to PSM-RF but the Observation Queue at Retire is full.
 - If PSM-A needs to read the destination register value for the retiring instruction while constructing a payload, but the PRF read ports are unavailable.

CHAPTER

3

PSM USE CASES

In this Chapter, we analyze several workloads from the SPEC benchmark suite to identify regions where the performance is poor on the baseline core. We focus on two primary microarchitectural constraints that plague many of these benchmarks: 1) high branch misprediction rate, and 2) high cache miss rate. We present source or assembly code of the identified regions of interest, where appropriate, to pinpoint the delinquent branch or load instructions and discuss the application behavior and the reason for low performance.

With the phase behavior of these workloads known, we develop application-specific custom branch predictor or custom prefetcher designs, to be synthesized on PSM-RE, to optimize and accelerate the regions of interest via microarchitectural PSM intervention and allow the baseline core to unlock higher performance. The design for each use case is discussed briefly before presenting results against normal baseline execution.

The main issue that arises when trying to develop and interface the designs in PSM-RF with the baseline core is the round-trip latency of communication between the faster core and the potentially slower PSM-RF. As an example of a possible design, consider an application which is constrained by a high branch misprediction rate. A customized branch predictor is synthesized in PSM-RF which can very accurately predict the direction of the delinquent branches. The core and PSM-RF then take the following actions when executing the application:

1. The fetch unit of the core fetches instructions as usual, but when the delinquent branch that has been identified as performing poorly with the default branch predictor is fetched, the core sends a request to the design synthesized in PSM-RF for a custom branch prediction.
2. The core stalls the fetching of instructions while waiting for a reply from PSM-RF.
3. The custom branch predictor designed in PSM-RF receives, after some communication delay, the request from the core for the custom branch prediction for the delinquent branch.
4. The custom branch predictor does some computations, or looks up its predictor tables, to generate the branch prediction for the delinquent branch and sends the prediction to the core.
5. The core receives the prediction from PSM-RF, after some communication delay, and uses it to override its own default prediction.
6. Instruction fetch by the core resumes, but all the above steps are repeated each time a dynamic instance of the delinquent branch is received.

The potential frequency difference between the core and PSM-RF, along with the latency of execution of the synthesized design on the slower PSM-RF, can result in 10s or 100s of

cycles of communication latency. Stalling the fetch unit for this duration each and every time the baseline core fetches a delinquent branch and needs a custom branch prediction, would negate any potential performance uplift of highly accurate custom branch prediction and can result in very poor performance.

The round-trip latency of communication necessitates development of *decoupled* PSM designs which can run ahead of the core’s instruction stream. Decoupled designs can *proactively* provide *timely* predictions to the core. As such, the core doesn’t stall waiting for the custom predictions from PSM-RF and can enjoy the benefits of higher branch prediction accuracy without getting adversely affected by the communication delays.

As a result, while the default predictors in the core have a requirement of accuracy, PSM designs introduce an additional requirement of *timeliness* for the custom predictors synthesized in PSM-RF. The use cases discussed in the sections below showcase the viability of developing decoupled PSM designs which satisfy these requirements and provide good performance.

3.1 Decoupled custom EXACT branch predictor

astar is a path-finding algorithm from the SPEC 2006 benchmark suite that works on indices in a 2-D graph. Profiling the *astar* benchmark reveals bottlenecks due to a high branch misprediction rate of its load-dependent branches. *astar*’s region of interest (ROI) is shown in Listings 3.1 and 3.2.

The *fill* function in Listing 3.1 calls the *makebound2* function in line 7, with an input and output worklist of indices (*bound1p* and *bound2p*). The store in line 3 provides the first index to the input worklist. The *makebound2* function processes the indices in the input worklist and populates the output worklist. For the next call to *makebound2*, the *fill* function switches the input and output worklists (line 10).

The *makebound2* function shown in Listing 3.2 pops indices from the input worklist

(line 7) and checks if its neighbors have already been visited. It uses a combination of the current index value and *yoffset* (line 3) to generate 8 neighboring indices, *index1* (lines 9, 18, 21-26), for each index in the input worklist. The *makebound2* function then checks if these generated indices were previously visited by looking up the *waymap* and *maparp* arrays (lines 10 and 11 for the first neighbor). If both, *waymap* and *maparp* branches are NT (not taken), it means that *index1* has not been visited yet. In this case, *index1* is marked as visited (line 14) and also stored in the output worklist (line 12) so that it can be worked on in the next call to *makebound2*.

```

1 bool wayobj::fill(...) {
2     ...
3     bound1p[0] = index(startx, starty);
4     ...
5     while((bound1 != 0) && (flend == false)){
6         if(flodd == false) {
7             bound1 = makebound2(bound1p, bound1, bound2p);
8             flodd = true;
9         } else {
10            bound1 = makebound2(bound2p, bound1, bound1p);
11            flodd = false;
12        }
13        ...
14    }
15 }

```

Listing 3.1: *astar*'s ROI in which PSM is enabled.

```

1 i32 wayobj::makebound2(i32pt bound1p, i32 bound1l,
    i32pt bound2p) {
2     ...

```

```

3  yoffset = maply;
4  ...
5  bound2l=0;
6  for(i = 0; i < bound1l; i++) {
7      index = bound1p[i];
8
9      index1 = index - yoffset - 1;
10     if(waymap[index1].fillnum != fillnum)
11         if(maparp[index1] == 0) {
12             bound2p[bound2l] = index1;
13             bound2l++;
14             waymap[index1].fillnum = fillnum;
15             ...
16         }
17
18     index1 = index - yoffset;
19     if(waymap[index1].fillnum != fillnum)
20         ...
21     index1=index-yoffset+1; ...
22     index1=index-1; ...
23     index1=index+1; ...
24     index1=index+yoffset-1; ...
25     index1=index+yoffset; ...
26     index1=index+yoffset+1; ...
27 }
28 }

```

Listing 3.2: The *makebound2* function with the load-dependent branches.

3.1.1 Analysis

The load-dependent branches (lines 10 and 11 in Listing 3.2) have a high branch misprediction rate which limits the IPC of *astar*. The conventional context used for branch prediction, PC and global branch history, fails to properly distinguish between the dynamic instances of these branches, thus performing poorly. Table 3.1 shows the misprediction rate and overall MPKI (Mispredictions Per Kilo Instructions) for these delinquent branches.

Table 3.1: Misprediction rate and MPKI for the delinquent branches in *astar*.

Branch	Misprediction rate	MPKI
<i>waymap</i>	23.01 %	22.52
<i>maparp</i>	20.66 %	9.30

With PSM’s ability to snoop the details of retiring instructions, coupled with construction of custom predictors, it is possible to use new/additional context not generally available at the fetch unit of the core. The proper context to predict the *waymap* and *maparp* branches would be the computed indices (*index1*). By snooping key information from the retiring instructions, PSM can facilitate *decoupled generation and use* of *index1* for predicting both these branches.

3.1.2 PSM design

The custom branch predictor developed in [Cha19], inspired from the EXACT [AOFR10] branch predictor’s active update mechanism, uses the *index1* context for predicting the delinquent branches in *astar*. [Cha19] shows a preliminary design where the custom branch predictor is tightly integrated with the core with no communication latency between the core and the custom branch predictor. The custom branch prediction lookups can be

done instantly and the tight integration with the core allows access to the partial-squash mechanism on a misprediction. This would be an example of a completely custom microarchitecture component that would benefit the specific *astar* workload immensely while not being used by any other application. This makes it a prime candidate for adoption as a PSM design where the custom branch predictor can be synthesized in the PSM-RF. We redesign, simplify, and adapt this custom design to work as a decoupled predictor while reducing the interactions needed with the core.

The region of interest (ROI) that is targeted is in Listing 3.1, from line 3 to 14. The instruction at the start of the ROI is tagged with 3 configuration bits: PSM Enable, Full Squash Mode, and Custom BP. The instruction at the end of the ROI is tagged with the configuration bit to disable PSM mode. This sets up the communication between the core and PSM-RF within the ROI and also asks the core to start consulting the PSM-Agent for custom branch predictions. The Full Squash Mode helps keep the synchronization of PSM-RF with the core, and the recovery of PSM-RF, simple at the expense of a higher branch misprediction penalty. The higher misprediction penalty shouldn't be an issue if the misprediction rate itself is low. The core sends messages to the PSM-RF for any misprediction squash events within the ROI.

The start of the *makebound2* function is tagged as a synchronization point (signalling the switching of the two worklists) in the Fetch Snoop Table (FST) to help in cases where the instruction stream in the core diverges from the expectations of PSM-RF, due to maybe resource constraints in the PSM-RF or prediction underflows to the core. The FST is setup to include PCs of the *waymap* and *maparp* branches so that their default predictions by the core can be overridden. The Retire Snoop Table (RST) is setup to snoop key instructions like the first index store (Listing 3.1, line 3), the *yoffset* value (Listing 3.2, line 3), and the actual direction of the targeted branches, when they retire. PSM-RF uses these snooped data to start decoupled generation of indices for the next worklist (once it has the first index) and processes them to make custom branch predictions.

Figure 3.1: Custom decoupled branch predictor design for *astar*.

Figure 3.1 shows the high-level design of our decoupled custom branch predictor for *astar*, configured in PSM-RF. The PSM design maintains two hardware structures (B and C) to mimic the input and output worklists. We keep two pointers for the worklists, *spec* and *arch* to allow decoupled execution while keeping recovery simple. The *arch* pointer points to the architecturally consistent ‘current index’ that is being worked on by the core and is kept in sync with the retire stream (albeit slightly delayed). The *spec* pointer, on the other hand, is used by PSM-RF to generate custom branch predictions in an independent decoupled fashion.

The first index in the input worklist is populated by snooping the store of first index (Listing 3.1, line 3). An index can be read (E) from the input worklist (using speculative pointer *spec*) to generate the computed indices (F) using the snooped *yoffset* value (Listing 3.2, line 3). These computed indices are used to look up simple direct-mapped predictors (G and H) for the *waymap* and *maparp* branches. The generated predictions are sent to PSM-A via the Intervention Queue at Fetch (IntvQ-F). If both *waymap* and *maparp* predictions were NT (not taken), the *waymap* table is actively updated (K) while maintaining an undo log (L).

The *maparp* table is updated passively (M) on mispredictions communicated by the retire stream. The actual branch directions are used to determine which indices are popped from the input worklist and pushed into the output worklist (J), while updating the *arch* pointer.

On a misprediction, when a squash message is received from the core, the *spec* pointer is reset to the *arch* pointer and the undo log is used to recover the *waymap* table (N). Since the *maparp* predictor table was passively updated, no recovery is needed.

The key idea here is that the *spec* pointer can run further ahead of the *arch* pointer (even far into the next worklist) and generate a custom branch prediction stream ahead of the core’s instruction stream. Use of the decoupled predictor, along with the proper context for prediction, *index1*, lets us generate accurate and timely predictions.

The custom branch predictions generated by PSM-RF might underflow if the worklist tables are insufficient to hold all generated indices. A SYNC command (tagged with the PC of start of *makebound2*) is pushed into the Intervention Queue at Fetch after an input worklist becomes empty. This allows the fetch unit to use default predictions until the next call to the *makebound2* function.

3.1.3 Evaluation Methodology

We briefly discuss the evaluation methodology and the different PSM parameters, below. This discussion applies to results presented in this, and other sections, for all the use cases presented in this work.

The baseline core configuration used in our studies is shown in Table 2.1. We use the top-weighted simpoint [SPHC02] (100 million instructions) of the SPEC benchmarks for our evaluation. The core runs applications compiled to the RISC-V ISA [WLP⁺14].

We use an in-house execution-driven cycle-level simulator to model the baseline core and the PSM design, along with different parameters in which the PSM design can vary. The key used for the different PSM parameters in the graphs for this and other sections, is described in Table 3.2.

Table 3.2: Key for different PSM parameters used in the sensitivity studies

Parameter	Description
clkC	C is the factor by which PSM-RF’s clock frequency is slower than that of the core ($C = \frac{f(CLK_{CORE})}{f(CLK_{PSM-RF})}$).
wW	W is the PSM superscalar width. PSM-RF can generate W predictions and push/pop W payloads into/from the communication queues, in a given CLK_{PSM-RF} cycle.
delayD	D is the pipelined execution latency of the design synthesized in PSM-RF, in CLK_{PSM-RF} cycles (e.g., for config clk4_w4, delay8 would mean 8 CLK_{PSM-RF} cycles or 32 CLK_{CORE} cycles).
queueQ	Q is size of the Observation and Intervention queues.
portP	P shows which PRF ports the PSM-Agent can contend on (e.g., portLS means PSM-A can opportunistically use PRF ports of the two load execution lanes in the baseline core, while LS1 means PSM-A is restricted to contending on only one LS port for PRF read).
chkptN	N is the number of CLK_{CORE} cycles it takes to checkpoint/restore the architectural state of the core.

3.1.4 Results

In this section, we show the results for the custom PSM design for predicting the delinquent branches in *astar*. While the baseline *astar* suffers an overall MPKI of ~ 32 , the PSM design is able to reduce the MPKI to ~ 2 , thus removing the branch misprediction bottleneck and yielding good performance improvement. Table 3.3 shows the reduction in misprediction rate for the targeted branches for one of the PSM configurations.

Figures 3.2, 3.3, and 3.4 show the performance of the decoupled PSM design over the baseline, for different PSM parameters. Figure 3.2 shows the sensitivity of the PSM design to the bandwidth between the core and PSM-RF. With a high value of C and a low value

Table 3.3: Misprediction rate for delinquent branches in *astar* for a PSM design (Configuration: clk4_w4, delay4, queue32, portALL with 32KB predictor tables, 512-entry worklists, and 32-entry undo log).

Branch	Baseline misprediction rate	PSM misprediction rate
<i>waymap</i>	23.01 %	0.63 %
<i>maparp</i>	20.66 %	2.64 %

of W , the communication bandwidth between the core and PSM-RF decreases. At low communication bandwidths, the fetch stage will stall while waiting for PSM-RF to generate and supply the predictions, thus limiting the performance gain or even causing slowdowns compared to baseline execution (e.g., configurations clk4_w1 and clk8_w1). The limitation of slower frequency of PSM-RF can be overcome by increasing the communication width. With sufficient bandwidth, the predictions generated by the custom branch prediction can reach the core in a timely fashion to provide good speedup (e.g., configurations clk4_w2 and clk4_w4).

Figure 3.3a shows that performance improvement reduces if the execution latency of the hardware synthesized on PSM-RF is too high. This is due to the increased initial delay in providing custom branch predictions before PSM-RF runs ahead and overtakes the core's instruction stream, as well as the high branch misprediction penalty of synchronizing the core and PSM-RF after a misprediction, via the handshaking protocol described in Section 2.2.1. The round-trip latency of sending the squash message, followed by the recovery of PSM-RF and an eventual acknowledgement received by the core, increases as the parameter D is increased, leading to higher stall time on each squash event.

Figure 3.3b shows that the performance is resistant to the size of the communication queues. This is because we are economical in snooping only key instructions that are crucial for starting the custom prediction engine. Because of this, for this use case, PRF port availability is also not an issue even if the opportunistic arbitration is statically reserved to

Figure 3.2: Performance of custom branch predictor for *astar*, for different C and W parameters. All configs are delay0, queue32, portALL, and have 32KB predictors, 512-entry worklists, and a 32-entry undo log.

Figure 3.3: Performance of custom branch predictor for *astar*, for different D and Q parameters. (a) All configs are clk4_w4 and queue32. (b) All configs are clk4_w4 and delay4. All configs in (a) and (b) are portALL, and have 32KB predictors, 512-entry worklists, and a 32-entry undo log.

only one PRF port as shown in Figure 3.4a.

The size of the undo log can limit how far the custom branch predictor in PSM-RF can

run ahead of the core's instruction stream. Figure 3.4b shows that a small size for the undo log is sufficient to provide enough run-ahead and is not a bottleneck.

Figure 3.4: Performance of custom branch predictor for *astar*, for different values of the P parameter and different number of entries in the undo log. (a) All configs are clk4_w4, delay4, queue32, with a 32-entry undo log. (b) All configs are clk4_w4, delay4, queue32, and portALL. All configs in (a) and (b) have 32KB predictors and 512-entry worklists.

Table 3.4: Misprediction rate for delinquent branches in *astar* for different predictor sizes.

Configuration	<i>waymap</i> misprediction rate	<i>maparp</i> misprediction rate
Baseline	23.01 %	20.66 %
32 KB	0.63 %	2.64 %
28 KB	0.95 %	4.43 %
24 KB	1.81 %	6.47 %
16 KB	6.07 %	12.11 %
12 KB	10.49 %	15.11 %

Figure 3.5: (a) Performance of custom branch predictor for *astar*, for different sizes of the predictor tables (all configs are clk4_w4, delay4, queue32, portALL, and have 512-entry worklists and a 32-entry undo log). (b) Performance of custom branch predictor for *astar*, for different number of entries in the worklist tables (all configs are clk4_w4, delay4, queue32, portALL, and have 32KB predictors).

Table 3.5: Misprediction rate for delinquent branches in *astar* for different worklist sizes.

Configuration	<i>waymap</i> misprediction rate	<i>maparp</i> misprediction rate
Baseline	23.01 %	20.66 %
512 entries	0.63 %	2.64 %
256 entries	1.46 %	3.29 %
224 entries	3.17 %	4.75 %
192 entries	8.17 %	9.12 %
160 entries	11.09 %	11.39 %
128 entries	14.60 %	14.31 %

Figure 3.5 shows the variation in performance improvement with different sizes of predictors and worklists. As the predictor size is reduced, the misprediction rate increases

Figure 3.6: Performance of a dedicated Oracle predictor for *astar* (configured for similar accuracy as the PSM design with 12KB predictors), for different values of the ‘restart delay’.

due to aliasing in our simple direct-mapped structures. While more sophisticated designs are possible, the results in Figure 3.5a help make an important observation. The adverse effects of the increased branch misprediction penalty (due to the required handshaking, along with full squash from head of ROB on mispredictions, and the latency of fixing the state in PSM-RF) when in PSM mode, makes the design much less tolerant to the misprediction rate compared to the baseline which does partial squashes (from middle of ROB). The performance of the PSM design falls below that of the baseline in high aliasing conditions (e.g., predictor size of 12KB in Figure 3.5a). Similar trends are seen when reducing the worklist sizes in Figure 3.5b. Tables 3.4 and 3.5 show the trend in the misprediction rate for the delinquent branches as the resources are constrained.

To understand the adverse effects of the full squash from ROB and the handshaking delay, we add a dedicated oracle predictor in the model which is configured to probabilistically achieve the accuracy of the PSM design with the 12 KB predictor (Table 3.4). The simulator

has a switch to force the core to do full squashes within the region of interest, or allow partial squashes (from middle of ROB), on a misprediction event. When the core is configured to do full squashes, an additional switch allows us to set the delay (in CLK_{CORE} cycles) after which the fetching of instructions can be started by the core ('restart delay'). This approximates the handshaking delay in the PSM design. The results for partial vs. full squash in the region of interest, as well as the sensitivity to the restart delay is shown in Figure 3.6. We see that using full squashes reduces the performance for a given accuracy compared to the performance for partial squashes, but the restarting delay further exacerbates this as its value increases.

3.2 Control Flow Decoupling

Control flow decoupling (CFD) [STR12] separates the branch slice from a branch's control-dependent instructions. The branch slice is executed first and the branch outcomes are pushed in an architectural Branch Queue (added to the ISA). The control-dependent instructions are then executed where the control flow at fetch is determined by popping precomputed branch outcomes from the Branch Queue.

Due to the decoupled nature of CFD, it is a natural candidate for implementation as a PSM design. A straight-forward implementation of CFD is possible in PSM where PSM-RF acts as a microarchitectural buffer for the precomputed branch outcomes (instead of an architectural Branch Queue). This is described below:

1. At the start of the identified ROI, PSM-A flushes the core and takes a checkpoint of the architectural state.
2. PSM-A turns on the 'Enable Instruction Fetch' mode where the core is directed to fetch instructions from PSM-A.
3. PSM-RF generates the instructions for the branch slice and injects it into the core for execution, via the intervention Queue at Fetch (IntvQ-F).

4. The core executes the branch slice. PSM-A snoops the computed branch outcomes or other relevant data, and sends it back to PSM-RF using the Observation Queue at Retire (ObsQ-R).
5. Once the branch slice is done executing, PSM-A restores the saved checkpoint and directs the core to use custom branch predictions sent by PSM-RF ('Custom BP' and 'Full Squash' mode) while the core resumes instruction fetch from the I-cache.
6. PSM-RF serves as the Branch Queue by storing the precomputed branch outcomes and streams the buffered branch outcomes as highly accurate branch predictions.

Depending on what kind of instructions are contained in the program slice injected into the code, other potential designs are also possible. For example, if PSM-RF only needs to read the state of the caches, it can generate the load addresses and send it as load OPs to the core using the Intervention Queue at Issue (IntvQ-IS). The loaded value can be observed via the Observation Queue at Execute (ObsQ-EX) and branch outcomes can be computed directly in PSM-RF. Injecting mini program slices into the core, however, provides a very general method of reading and changing the microarchitectural state of the core as well as the caches and can benefit from the performance of a superscalar OOO core for executing its instructions.

While it is simple to implement CFD via the methods enumerated above, PSM offers additional opportunities to target even the inseparable branches (which are not CFD-friendly) by keeping additional structures in PSM-RF to fix the control-flow dynamically. We use *astar* to show this use case.

3.2.1 Analysis

Listing 3.3 shows the region of interest in the *astar* benchmark that is targeted for Control Flow Decoupling. As mentioned in Section 3.1.1, the *waymap* and *maparp* branches (lines 7 and 8) have a high-misprediction rate and limit the IPC potential of the baseline core.

The objective of CFD is to separate and pre-execute these branches for the entire *bound11* loop. However, the store to *waymap* in line 10 makes the control flow itself dependent on its control-dependent instructions, making a simple implementation of CFD difficult. PSM, however, allows a naive separation of these branches while keeping an additional direct-mapped structure (the *index table*) to fix the generated predictions on the fly.

```

1 i32 wayobj::makebound2(...) {
2     ...
3     bound2l=0;
4     for(i = 0; i < bound1l; i++) {
5         index = bound1p[i];
6         index1 = index - yoffset - 1;
7         if(waymap[index1].fillnum != fillnum)
8             if(maparp[index1] == 0) {
9             ...
10            waymap[index1].fillnum = fillnum;
11            ...
12        }
13        ...
14    }
15    ...
16 }

```

Listing 3.3: *astar*'s ROI targeted for Control Flow Decoupling.

3.2.2 PSM design

The RST is setup to checkpoint the architectural state before the *bound11* loop (line 4), and start the ‘Enable Instruction Fetch’ mode. PSM-RF then streams the instructions of the CFD Program Slice (shown in Listing 3.4) into the core. The RST is setup to snoop the

values of instructions in lines 1, 3, 5, and 6. These snooped values are then used by PSM-RF to generate the predictions. A simple direct-mapped index table is kept in PSM-RF to fix the computed control flow. If both *waymap* and *maparp* branches are computed as NT (not-taken), the corresponding index entry is set. After snooping the *waymap* load (line 5), the index table is consulted: if the corresponding entry is already set, the *waymap* direction is forced ‘taken’ in the buffered branch outcomes. The index table is cleared at the start of every CFD phase. Once the CFD branch slice is done executing, the core is switched back to fetching instructions from the I-Cache and directed to take overriding custom predictions for the *waymap* and *maparp* branches from the PSM-Agent.

```

1 LOAD(fillnum);
2 for(i=0; i<bound11; i++){
3     index = bound1p[i];
4     index1 = index-yoffset-1;
5     LOAD(waymap[index1].fillnum);
6     LOAD(maparp[index1]);
7     ...
8 }
```

Listing 3.4: Branch slice injected into the core for Control Flow Decoupling.

Using the index table allows generation of accurate branch outcomes for *astar* even in the presence of an inseparable control flow. At the same time, pregeneration and buffering of the branch outcomes allows the custom branch prediction stream to be ahead of the core’s instruction stream, thus generating timely branch predictions.

Implementing CFD as one of the PSM use cases, instead of as a dedicated design added to the baseline core, has an overhead of duplicate execution of the branch slice: once as part of the CFD slice, and once as part of normal baseline execution, when in contrast, a dedicated design would only need to execute this branch slice once. We can still get good performance improvement with CFD as a use case of PSM if the benchmark, like *astar*,

suffers from a severe branch misprediction bottleneck.

3.2.3 Results

CFD reduces the overall MPKI of *astar* from ~32 to 0.23. Table 3.6 shows the misprediction rate for the targeted delinquent branches without and with CFD. We see that CFD enables very accurate branch prediction.

Table 3.6: Misprediction rate for CFD-targeted branches in *astar*.

Branch	Baseline misprediction rate	CFD misprediction rate
<i>waymap</i>	23.01 %	0.05 %
<i>maparp</i>	20.66 %	0.01 %

The performance results for the different PSM parameters are shown in Figures 3.7, 3.8, and 3.9. We see that while this PSM use case is resistant to several PSM parameters, the performance is very sensitive to the bandwidth between the core and PSM-RF (Figure 3.7). This is because at low bandwidths, the injected CFD slice itself becomes a bottleneck due to low instruction fetch rate from PSM-RF.

Due to the very low MPKI, performance is very resistant to the delay of the PSM design, as shown in Figure 3.8a. Since the CFD mode is entered infrequently, checkpointing delay has limited effect on performance (Figure 3.9a). Figure 3.9b shows that, even with heavy snooping of the load instructions in the tight branch slice loop (Listing 3.4), when in CFD mode, the performance gets substantially affected only if the number of ports that PSM-Agent can arbitrate on is severely reduced. Even with just two assigned ports, ample opportunities to opportunistically read the register values from PRF exist.

Figure 3.7: Performance of Control Flow Decoupling, for different C and W parameters. All configs are delay0, queue32, chkpt32, portALL, with an 8KB index table. Legend available in Section 3.1.3.

Figure 3.8: Performance of CFD, for different D and Q parameters. (a) All configs are clk4_w4, queue32, chkpt32, portALL. (b) All configs are clk4_w4, delay4, chkpt32, portALL. All configs in (a) and (b) use an 8KB index table. Legend available in Section 3.1.3.

Loop Buffer Optimization

The bandwidth sensitivity observed in Figure 3.7 can be optimized by adding a loop buffer in the PSM-Agent to record the instructions received from PSM-RF and replay it in the core directly without further communication with the PSM-RF. Two new intervention

Figure 3.9: Performance of CFD, for different N and P parameters. (a) All configs are clk4_w4, delay4, queue32, portALL. (b) All configs are clk4_w4, delay4, queue32, chkpt32. All configs in (a) and (b) use an 8KB index table. Legend available in Section 3.1.3.

commands can be added: 1) LOOP_RECORD, which signifies the start of one iteration of the loop instructions, and 2) LOOP_END (along with a count of how many times the loop should be iterated) which signals the last instruction in a loop slice.

With this optimization, the PSM-RF can send the branch-slice loop instructions to the PSM-Agent only once (one iteration). The fetch unit of the core can then get instructions directly from the loop buffer in PSM-Agent for further iterations, thus mitigating the issue of low instruction fetch bandwidth from PSM-RF.

Figure 3.10 shows the performance results for the CFD use case, for different communication bandwidths, in the presence of a loop buffer. Comparing this with the performance of CFD in Figure 3.7 clearly shows the benefits of including an instruction buffer in the PSM interface. The performance improvement is also more resistant to the varying communication bandwidth, than without the loop buffer. Figures 3.11 and 3.12 show that the performance is resistant to the other PSM parameters, and that providing support for arbitrating on only few PRF ports is sufficient to get most of the performance potential of CFD.

The speedup for *astar* using CFD is lower than the fully customized branch predictor

presented in Section 3.1 due to the overhead of executing the branch slice. The overhead is especially high for the branch slice of our use case to the high number of loads, for which execution quickly gets resource-constrained by the limited size of the load queue in the core.

Figure 3.10: Performance of Control Flow Decoupling (with loop buffer), for different C and W parameters. All configs are delay0, queue32, chkpt32, portALL, with an 8KB index table. Legend available in Section 3.1.3.

The accuracy of the CFD branch predictor suffers if the size of the index table kept to dynamically fix the branch outcomes for the inseparable delinquent branches of *astar*, reduces and starts aliasing. Figure 3.13 shows the performance sensitivity for different sizes of the index table, while Table 3.7 shows the trend in the misprediction rate. Since aliasing in the index table only affects the *waymap* branch, accuracy of the *maparp* branch remains unaffected.

Figure 3.11: Performance of CFD (with loop buffer), for different D and Q parameters. (a) All configs are clk4_w4, queue32, chkpt32, portALL. (b) All configs are clk4_w4, delay4, chkpt32, portALL. All configs in (a) and (b) use an 8KB index table. Legend available in Section 3.1.3.

Figure 3.12: Performance of CFD (with loop buffer), for different N and P parameters. (a) All configs are clk4_w4, delay4, queue32, portALL. (b) All configs are clk4_w4, delay4, queue32, chkpt32. All configs in (a) and (b) use an 8KB index table. Legend available in Section 3.1.3.

3.3 Custom Branch Predictor for *wrf*

In this section, we showcase a custom branch predictor for the *wrf* benchmark from SPEC 2017 benchmark suite. Unlike the branch predictors presented in previous sections, the branch outcomes for the delinquent branch in *wrf* can be generated computationally, once we have snooped some key data from the retire stream, in the beginning of the region of the

Figure 3.13: Performance of CFD (with loop buffer) for different sizes of the index table. All configs are clk4_w4, delay4, queue32, chkpt32, portALL). Legend available in Section 3.1.3.

Table 3.7: Misprediction rate for delinquent branches in the CFD use case for different index table sizes.

Configuration	<i>waymap</i> misprediction rate	<i>maparp</i> misprediction rate
Baseline	23.01 %	20.66 %
8 KB	0.05 %	0.01 %
4 KB	0.26 %	0.01 %
2 KB	0.90 %	0.01 %
1 KB	2.14 %	0.01 %

interest. No hardware predictor tables are needed and thus there are no issues with resource constraints and aliasing. A customized Finite State Machine (FSM) is designed, which, once started, runs ahead and generates accurate predictions for the targeted delinquent branch.

```

1  105a36c:  lw      a5, 28(sp)
2  105a370:  lw      a2, 44(sp)
3  105a374:  subw    a4, a4, a3
4  105a378:  sw      a4, 24(sp)
5  105a37c:  bleu    a2, a5, 105a38c

```

```

6    105a380:    addiw    a4,a4,-1
7    105a384:    slliw    a5,a5,0x1
8    105a388:    sw       a4,24(sp)
9    105a38c:    li       a4,31
10   105a390:    lui      a3,0x40000
11   105a394:    li       a1,0
12   105a398:    addiw    a4,a4,-1
13   105a39c:    bltu     a5,a2,105a3a8
14   105a3a0:    or       a1,a1,a3
15   105a3a4:    subw     a5,a5,a2
16   105a3a8:    srlw     a3,a3,0x1
17   105a3ac:    slliw    a5,a5,0x1
18   105a3b0:    bnez     a4,105a398
19   105a3b4:    andi     a3,a1,127
20   105a3b8:    li       a4,64
21   105a3bc:    beq      a3,a4,105a3e4
22   105a3c0:    sw       a1,28(sp)
23   105a3c4:    j        105a300

```

Listing 3.5: Region of interest in the *wrf* benchmark.

3.3.1 Analysis

Listing 3.5 shows the assembly code of the region suffering from branch misprediction bottlenecks that was targeted for this use case. The delinquent branch is in line 13, and is part of a loop (lines 12 to 18) that has a fixed number of iterations (line 9). The delinquent branch is dependent on two registers: *a2* and *a5*. Register *a2* is initialized in line 2 and remains unchanged during the loop execution. Register *a5*, on the other hand, gets modified within the branch’s control dependent region (lines 15 and 17) and introduces an unpredictable

pattern in the delinquent branch. Register *a5* gets initialized in line 1 and is sometimes updated before the loop, in line 7, depending on the outcome of the branch in line 5. The branch in line 5 itself has somewhat low predictability, and even though this branch instance is infrequent because it is outside the loop, since it affects the updates to register *a5*, we predict this branch as well in our PSM design. The branch in line 5 is dependent only on the initial values of registers *a5* and *a2*. The misprediction rate of the branches targeted by our custom branch predictor is shown in Table 3.8.

Table 3.8: Misprediction rate and MPKI of the delinquent branches in *wrf*.

Branch (refer Listing 3.5)	Misprediction Rate	MPKI
105a39c (line 13)	35.41 %	22.65
105a37c (line 5)	14.27 %	0.30

3.3.2 PSM design

The region of interest shown in Listing 3.5 is marked for PSM intervention via custom branch prediction. The values of registers *a5* and *a2* are snooped from the instructions in lines 1 and 2. These are sufficient to start the designed FSM and generate the custom branch predictions in a decoupled manner. The FSM models the updates to the *a5* register and makes extremely accurate predictions which are sent to the core via the Intervention Queue at Fetch.

3.3.3 Results

The targeted region of interest accounts for ~40 % of the total execution time, on the baseline core, for the *wrf* simpoint used in our study. Figures 3.14 through 3.16 show

the performance potential of the PSM design compared to the baseline. Due to the tight loop that the delinquent branch resides in, the performance becomes sensitive to low bandwidths and high delays as shown in Figures 3.14 and 3.15, respectively. Performance is resistant to the other PSM parameters.

Figure 3.14: Performance of the PSM design for *wrf*, for different *C* and *W* parameters. All configs are delay0, queue32, portALL. Legend available in Section 3.1.3.

Figure 3.15: Performance of the PSM design for *wrf*, for different *D* parameters. All configs are clk4_w4, queue32, portALL. Legend available in Section 3.1.3.

Figure 3.16: Performance of the PSM design for *wrf*, for different Q and P parameters. (a) All configs are `clk4_w4`, `delay4`, `portALL`. (b) All configs are `clk4_w4`, `delay4`, `queue32`. Legend available in Section 3.1.3.

3.4 Load Prefetching

PSM can be used to target applications which have performance bottlenecks due to high cache miss rates. Prefetching is a natural candidate for PSM as it already has requirements of both accuracy and timeliness. Design of a custom prefetch generation engine, however, has two advantages:

- Application-customization allows proactive generation of extremely accurate prefetches for the delinquent loads.
- With PSM’s ability to snoop the retire stream, it can keep track of the core’s position in the instruction stream and, if needed, adaptively update the prefetch distance to provide good timeliness.

Figure 3.17 shows a high-level overview of the design adopted for the use cases in this section. The ‘Prefetch Generation Engine’ is the customized prefetcher for a targeted workload. Key data from the core’s retire stream is snooped and fed into the Prefetch

Figure 3.17: PSM design for load prefetching.

Generation Engine, which uses this information to generate the prefetch OPs that are sent to the core via the Intervention Queue at Issue (IntvQ-IS). The core executes the received prefetch OPs from the head of IntvQ-IS when it finds a bubble in the load execution lanes (Section 2.2.5). The Prefetch Generation Engine can be as simple, or as complex, as needed by the targeted application.

The optional ‘Adaptive Prefetch Distance’ block also snoops the payloads popped from the Observation Queue. Taking advantage of the access to the Retire stream, it can, for example, implement a simple sampling-based performance feedback mechanism to adaptively update the prefetch distance to achieve better timeliness. This block can periodically track a proxy IPC, for a given prefetch distance, and update the prefetch distance, if needed, to achieve better performance. Some of the benchmarks analyzed in this section have delinquent loads in small tight loops which makes prefetching, using PSM, sensitive to the bandwidth and delay between the core and PSM-RF. For these benchmarks, this relatively simple adaptive policy works quite well in finding an optimal prefetch distance, although other more complex policies (for example, MLP-awareness) can be implemented if needed by any specific application.

We demonstrate the prefetching use case here with the help of the *libquantum* benchmark (Section 3.4.1) from the SPEC 2006 benchmark suite. *libquantum*’s simplicity allows

focusing on the fundamental ideas of PSM for prefetching. Custom prefetchers were also developed for other benchmarks from the SPEC 2006 benchmark suite. The overall structure remains similar to the design shown in Figure 3.17, but the Prefetch Generation Engine is customized according to the needs of the specific benchmark. We briefly discuss the other benchmarks that were also targeted for load prefetching in Sections 3.4.2 through 3.4.5, before presenting results in Section 3.4.6. Shubham Bhawalkar collaborated in the design of the prefetcher state machines for *lbm*, *bwaves*, and *leslie* benchmarks, as part of his Master’s thesis.

3.4.1 *libquantum*

```

1 for(i=0; i<reg->size; i++)
2 {
3     if(reg->node[i].state & ((MAX_UNSIGNED) 1 << control1))
4     {
5         if(reg->node[i].state & ((MAX_UNSIGNED) 1 << control2))
6         {
7             reg->node[i].state ^= ((MAX_UNSIGNED) 1 << target);
8         }
9     }
10 }

```

Listing 3.6: *libquantum*’s quantum_toffoli function

libquantum has two delinquent loads, one each in the *quantum_toffoli* and *quantum_sigma_x* functions. Listing 3.6 shows the *quantum_toffoli* function where the delinquent load is in line 3 (`reg->node[i].state`). The baseline VLDP prefetcher, though accurate for the simple constant-stride pattern seen in *libquantum*, is not timely enough to hide the high memory access latency. The delinquent load misses in several levels of the cache hierarchy and causes low performance. The *quantum_sigma_x* function is very similar and

also bottlenecks the performance.

The customized prefetcher design for *libquantum* targets the delinquent loads in both the ROIs. The start and end of both ROIs are recorded in the Retire Snoop Table to enable and disable PSM intervention, respectively. For the *quantum_toffoli* function, start of the loop in line 1 (Listing 3.6) is marked for enabling PSM ('Enable PSM Mode' flag in the RST entry). PSM-A also snoops the load in line 3 and iteration count (`reg->size`) in line 1, and sends these values to PSM-RF.

With the knowledge of the base address for the delinquent load and the final iteration count of the loop, snooped from the retiring instructions, along with the value of the stride obtained by inspecting the assembly, we can build a simple custom Finite State Machine ('Prefetch Generation Engine') which generates very accurate prefetch OPs.

The snoop of the load in line 3 also helps keep track of the loop iteration that the core is working on. The rate at which this payload is received from ObsQ-R helps keep track of a proxy IPC which is sampled by the 'Adaptive Prefetch Distance' block to adjust the prefetch distance.

3.4.2 *lbm*

lbm has a bottleneck in its *LBM_performStreamCollide* function due to 6 delinquent loads which access different cache lines. These loads are clustered and need to be prefetched together to ensure good performance. The VLDP prefetcher is unaware of the MLP (Memory-Level-Parallelism) implications of the clustered loads and shows very uneven latency reduction (Figure 3.18). This simply shifts the bottleneck from one delinquent load to the other, with very limited performance benefit. Addition of an L1 prefetcher helps mitigate this to some extent, and further reduces the average latency of the delinquent loads. However, issues with timeliness of prefetching still leave a performance gap compared to an idealistic baseline with perfect data cache (in which the memory accesses always hit).

The custom prefetcher engine designed for *lbm* is cognizant of the clustered nature of

Figure 3.18: Uneven latency reduction for the delinquent loads in *lbm*, by the VLDP prefetcher.

the loads. The prefetch OPs for all delinquent loads in the cluster are generated as a set, and either pushed into the IntvQ-IS, or dropped, together. This MLP-awareness allows us to get good performance.

3.4.3 *bwaves*

The delinquent loads in *bwaves*' *mat_times_vec* function have a complex multi-stride pattern and are hidden within several layers of nested loops. Development of a completely customized finite state machine is necessary to generate accurate prefetch OPs.

3.4.4 *milc*

The *uncompress_anti_hermitian* function contains the cluster of delinquent loads which span multiple cache lines. This function is called repeatedly from a loop in the *add_force_to_mom* function. The behavior of both these functions is modeled using a customized Finite State Machine to generate accurate prefetches.

3.4.5 *leslie*

The top-weighted simpoint for *leslie* has several small regions with high cache miss rate, with the delinquent loads inside several layers of loop nesting, similar to *bwaves*. Finite state machines targeting three of these regions (one ROI in function *EXTRAPK* and two ROIs in function *FLUXJ*) were developed for the results presented in Section 3.4.6.

3.4.6 Results

This section presents the performance results when using PSM for prefetching delinquent loads in five benchmarks from the SPEC 2006 suite. Each benchmark uses its own customized Prefetch Generation Engine and is able to achieve good performance improvement over baseline. For example, for the *lbm* simpoint used in our study, the region of interest spans the entire 100 million instructions, and while the baseline prefetchers together were able to reduce the average latency of the delinquent loads by ~75 %, the customized prefetcher design synthesized in PSM-RF is able to convert the misses to almost all hits and approaches the performance of a baseline core with a perfect data cache.

The performance is very resistant to the bandwidth between the core and PSM-RF (Figure 3.19), as well as the execution latency of PSM-RF (Figure 3.20). This is because of two reasons: 1) unlike the custom branch prediction use cases, for prefetching, the baseline core execution doesn't stall waiting for payloads from PSM-RF, and 2) for benchmarks which have delinquent loads in small tight loops, the 'Adaptive Prefetch Distance' block is used to help tolerate a range of bandwidth and delay values by adjusting the prefetch distance dynamically. For example, for *libquantum*, the average prefetch distance (across all invocations of the PSM mode, starting with an initial prefetch distance of 60) chosen by the 'Adaptive Prefetch Distance' block is ~180 for the *clk4_w4* configuration, while the average prefetch distance increases to ~310 when bandwidth reduces in the *clk4_w1* configuration.

Figure 3.21 shows that the performance is resistant to the different sizes of the commu-

Figure 3.19: Performance of PSM for load prefetching, for different C and W parameters. All configs are delay0, queue32, portALL. Legend available in Section 3.1.3.

nication queues, while Figure 3.22 shows that PRF port availability is not an issue for these use cases even if port arbitration is limited to one or two pre-assigned ports to simplify the hardware. Finally, Figure 3.23 compares the performance of a PSM design when using an adaptive prefetch distance, against different prefetch distance values that were statically fixed.

Figure 3.20: Performance of PSM for load prefetching, for different D parameters. All configs are clk4_w4, queue32, portALL. Legend available in Section 3.1.3.

Figure 3.21: Performance of PSM for load prefetching, for different Q parameters. All configs are clk4_w4, delay4, portALL. Legend available in Section 3.1.3.

3.5 Load-dependent Load Prefetching

Applications with load-dependent loads which miss in the cache hierarchy are difficult to prefetch with conventional prefetchers because of their irregular access pattern. A customized PSM design can be developed which can target the prefetching of load-dependent

Figure 3.22: Performance of PSM for load prefetching, for different P parameters. All configs are clk4_w4, delay4, queue32. Legend available in Section 3.1.3.

Figure 3.23: Performance of PSM for *libquantum*, with an adaptive prefetch distance (starting from an initial distance of 60), compared to static prefetch distances. Configuration is clk4_w4, delay0, queue32, portALL. Legend available in Section 3.1.3.

loads (e.g., pointer chasing). We show this use case using the *mcf* benchmark from the SPEC 2017 benchmark suite.

Listing 3.7 shows the region of interest within the *primal_bea_mpp* function of *mcf* which suffers from a low performance due to five delinquent loads: `arc->ident` (line 8),

`arc->tail` (line 10), `arc->head` (line 11), and two load-dependent loads of the potential variable (lines 10 and 11). The loads for the `tail` and `head` variables access the same cache line as that for `ident`, so prefetching for `ident` takes care of the other two loads. The load for `ident` has a constant stride and can be prefetched in a manner similar to *libquantum*, after snooping the base address (line 1), the end address (lines 3 and 5), and the address increment (line 7) for the `arc` variable. On the other hand, the loads for the potential variable are dependent on loads for `tail` and `head`, thus generating an irregular address stream which can't be computationally determined.

```

1 arc = *end_arc + step;
2 if (*end_arc >= full_group_end_arc)
3     *end_arc = *end_arc + max_elems - 1;
4 else
5     *end_arc = *end_arc + max_elems;
6
7 for ( ; arc < *end_arc; arc += num_threads) {
8     if( arc->ident > BASIC)
9     {
10        red_cost = arc->cost - arc->tail->potential
11                                + arc->head->potential;
12        ...
13    }
14 }
```

Listing 3.7: *mcf*'s ROI with the delinquent loads.

However, the interface supported by the core and PSM-A allows sending load OPs to the Intervention Queue at Issue (IntvQ-IS) which are executed by the core on a bubble in the load execution lanes (Section 2.2.5). The loaded value is then sent back to PSM-RF via the Observation Queue at Execute (ObsQ-EX). This mechanism can be used to do

prefetching for pointer-chasing applications. In the case of *mcf*, as shown in Figure 3.24, PSM-RF generates and issues load OPs for the `tail` and `head` variables ('Load Stream'). The values returned by the core are packaged as the addresses for prefetch OPs, for the potential variables, which are then sent to the IntvQ-IS.

Due to the structural hazard of being able to execute only one load OP from the head of IntvQ-IS, the Load Stream might get slowed down if a load OP misses in the cache and gets stuck at the head of IntvQ-IS. To avoid this, the 'Prefetch Stream' for load of `ident` (loads for `tail` and `head` fall within the same cache line) runs ahead (by the 'Load-delay Distance') and prefetches the cache line so that IntvQ-IS is not blocked by the missing load OPs from the 'Load Stream'.

Figure 3.24: Load-dependent load prefetching for *mcf*.

3.5.1 Results

The *primal_bea_mpp* function targeted for *mcf* accounts for ~30% of the total execution time in the baseline. Figures 3.25 and 3.26 show that the PSM design achieves ~18% performance

improvement while being very resistant to variations in different PSM parameters, similar to the load prefetching use cases in Section 3.4.

Figure 3.25: Performance of the PSM design for *mcf*, with different C and W parameters. All configs are delay0, queue32, portALL. Legend available in Section 3.1.3.

Figure 3.26: Performance of the PSM design for *mcf*, with different D , Q , and P parameters. (a) All configs are delay0, queue32, portALL. (b) All configs are clk4_w4, delay4, portALL. (c) All configs are clk4_w4, delay4, queue32. Legend available in Section 3.1.3.

CHAPTER

4

DISCUSSION

In this chapter, we discuss several high-level ideas related to Post-Silicon Microarchitecture. We discuss issues related to supporting multiprocessing as well as other types of use cases not presented in Chapter 3. The choice of reconfigurable logic for PSM-RF is another aspect which can potentially have far-reaching effect on the fundamental trade-offs in the design of the PSM interface and the kinds of use cases that can be supported. We discuss how the current iteration of the PSM-Agent is tied closely to the assembly instructions of the targeted executable, the issues that arise due to this, and a possible mitigation.

4.1 Multi-programming

Previous proposals which use reconfigurable logic to implement accelerators for applications, typically have architecturally important data buffered in their programmable fabric. As such, in a multi-programming environment, the architectural state of the reconfigurable components needs to be saved and restored for each context switch. Unlike these proposals, PSM targets the microarchitectural constraints in an application. In our PSM design, the reconfigurable fabric (PSM-RF) doesn't keep any architecturally important data. So, no state needs to be saved from or restored to PSM-RF, on a context-switch, for correctness. This falls under the 'Accelerator-as-a-Predictor' paradigm where while accuracy of the predictors built in PSM-RF is important, correctness is not a requirement.

However, for good performance, it is still desirable to preserve the predictor states in PSM-RF for when the application resumes execution after a context-switch. A simple solution for this would be to view PSM-RF as an Operating System (OS) resource. The applications request the OS for access to the PSM resources, and the OS can give an exclusive 'PSM-use' permission to one application based on some prioritization policy. The OS disables the PSM-Agent (snoop tables, intervention and observation queues) if the application is switched out, without squashing the PSM state. The PSM interface in this work has been designed in such a way that any application can continue its baseline execution on the processor core without any knowledge of whether or not a reconfigurable fabric is present. PSM transparently intervenes in the microarchitectural execution, if enabled for an application, or allows normal baseline execution, if disabled.

An alternative would be to allow partitioning the PSM-RF among multiple processes, coupled with partial reconfiguration and tagging of each communication payload with the process IDs. This is similar to the designs in [WA11] and [WCA08], where a pool of programmable fabric is divided among the executing threads, as needed. This alternative would still contend with issues of state save and restore if the number of processes, that need access to the reconfigurable logic, becomes more than the allowed number of logic

partitions.

4.2 Choice of Reconfigurable Fabric

There are several choices available for the type of the reconfigurable logic used in PSM-RF. These are itemized below.

- Fine-grained reconfigurable fabric in the style of FPGAs offer great flexibility at the expense of clock frequency.
- Coarse-grained reconfigurable architectures (CGRAs) offer a balance between performance and configuration.
- A Custom Reconfigurable Fabric can also be developed for our use cases since they only need some RAM and few configurable ALUs and FSM blocks.

The choice of reconfigurable fabric used in the PSM design greatly affects a number of parameters (clock frequency of PSM-RF, latency of execution, energy, etc.) which might facilitate or exclude certain use cases. Most importantly, the choice of reconfigurable fabric would affect the requirement of timeliness for the developed use cases. For example, for custom branch prediction, CGRAs or custom logic would reduce the requirement of how far ahead of the core the decoupled design in PSM-RF needs to run to provide timely predictions, compared to an FPGA-style PSM-RF. Due to the varied possibilities of reconfigurable fabrics, we keep our model for PSM-RF very general and offer results for various parameters of frequencies and delays.

4.3 Programming & source-code recompilation

Since PSM is an application-level optimization, we believe it is ultimately the responsibility of the application developer (maybe working in conjunction with a hardware engineer) to:

1) profile the workload (and the assembly) to find the regions of interest, and 2) program the FST and RST in the PSM-Agent along with the design of the components in PSM-RF. One potential issue with this is that the PSM-Agent would be tied closely to the assembly of the executable targeted for PSM intervention.

The PSM interface presented in this dissertation was developed from the perspective of PSM-RF. The application development and baseline execution can continue without any knowledge of the presence of an intervening PSM design. The snoop tables in PSM-A record PCs of instructions that are targeted for branch prediction, or are snooped for key information at retire time. Since the PCs might change on recompilation of the source code, earlier recorded PCs might become stale. One extreme option is to move the interface closer to the ISA and the source code so that a programmer can encode the details of FST or RST fields in the instructions directly via macros or pragmas similar to that for the "likely" bit used for providing branch prediction information to the hardware. However, this raises issues of transparency of the PSM fabric and whether the application (with the PSM-specific options coded directly in the instructions) would be able to run on a core without a PSM-RF coupled to it. This would potentially also require subsequent updates to the ISA as the PSM interface is updated. A middle ground would be to use the programmer-inserted macros to automatically generate and add PCs of required instructions into a separate configuration bitstream, which is then used to populate the FST and RST without needing any changes in the assembly or the instruction encodings in the ISA. A potential future work would be to look at these alternatives in more detail.

4.4 Other use cases & updates to the PSM Interface

The use cases presented in this dissertation focused on custom conditional branch predictors and custom prefetchers. Ideally, the reconfigurable fabric should function as a virtual hardware that could be customized into any kind of desired microarchitecture compo-

nent. Some examples of enhancements would be application-customized value predictors, indirect branch predictors, memory-dependence predictors, etc. However, due to the potentially high round-trip latency of communication between the core and the PSM-RF, and limitations stemming from the current design of the PSM-Agent (in an effort to keep PSM-A simple and non-intrusive), many of these microarchitectural ideas might not be feasible as a PSM design.

As a specific example of when a use case might not work, we look at a possible design for custom value prediction. Value Predictors typically have three logical parts - prediction, update, and verification. An application-specific value predictor structure could be designed in the PSM-RF that generates the predictions to be sent to the core. An explanation of how the core and the PSM-RF can be interfaced for this use case, and the issues that arise, is described below:

- **Prediction:** The custom value predictor in PSM-RF generates a predicted value and wants to send it to the core's front-end. This is similar to how custom branch predictions are supplied to the core and can be accomplished in the same way by using commands through the Intervention Queue at Fetch. The core would, however, have to provide support to add this predicted value in its PRF and allow the dependent instructions to use it.
- **Update:** The predictor tables in PSM-RF can be kept updated using the values snooped from the retire stream. This is similar to the other use cases.
- **Verification:** Since the core doesn't provide a dedicated verification machinery, it would need to communicate again with the PSM-RF (when the instruction that was predicted is finally executed) to verify the accuracy of prediction. The predicted instruction can't be retired until its verification is complete. This will need to be done for every instruction that was value predicted. This can severely hinder the retire rate of the core and reduce performance compared to that for the baseline core.

Our efforts to develop a value prediction use case based on CAVA [CST⁺06] didn't show any performance improvement because of the multiple round-trips of communication needed to predict and verify the values. This is in contrast to the use case of branch prediction where only 1 round-trip communication is needed. The difference is that while the baseline core provides extensive support for branch prediction and verification, similar support for value prediction is lacking. If the baseline core were to provide a mechanism of using and verifying the value predictions, then custom Value Predictors could become a viable PSM use case with only minor changes in the PSM interface. In light of this, it is important to note that the PSM interface presented in this work is not claimed to be general enough to support any and all use cases that one can think of. Instead, it is designed minimally to not be intrusive to the core, and still support the use cases that are most viable, and advantageous, for the current microprocessors.

4.5 Automation

Automated development of some of the use cases presented in Chapter 3 is possible. For example, in [STR12], Sheikh et. al. develop compiler techniques to identify the branches of interest for Control Flow Decoupling (CFD), and separate the branch slice from its control-dependent region. This can be leveraged for an automated design of the CFD use cases. However, for inseparable branches in benchmarks like *astar*, which require a custom mechanism to fix the generated predictions dynamically, automation would be difficult. Similarly, for completely customized designs such as the custom EXACT branch predictor in Section 3.1, automation might not be an option. On the other hand, regarding the prefetching use cases, [CHA⁺15] presents a hardware-based iterative algorithm to determine a load's backward slice. With PSM's ability to snoop the retiring instructions, it would be feasible to include this in the PSM design for prefetchers. Compiler-based solutions, coupled with High Level Synthesis [CM10] of the prefetching finite state machines, might also be

viable. However, the crux of the matter is that automation of the PSM design is highly dependent on how customized, for the targeted application and feature, we want our PSM design to be. A need for high customization would lead to a lower probability of automation, and vice versa.

CHAPTER

5

RELATED WORK

Similar design ideas of integrating reconfigurable hardware with a core, to accelerate the performance of superscalar processors, have been explored in the past. We discuss these related works in Section 5.1. Another set of related works stem from the dedicated designs that were proposed as microarchitectural enhancements for superscalar cores and have been adapted in this work for showcasing some of the use cases of PSM. We discuss these in Section 5.2.

5.1 Reconfigurable fabric for acceleration

There have been several proposals which explore integrating a reconfigurable fabric on the same chip as a high-performance core. While they differ in how tightly the fabric

is coupled to the core, as well as on the granularity of the targeted program regions, all previous work focused on accelerating the *computation* of hot loops, traces, or sets of instructions. The limitations of the core were circumvented by dynamically synthesizing a faster or more efficient application-customized accelerator in the reconfigurable logic. PSM, on the other hand, targets the *microarchitectural* inefficiencies in a core’s execution to allow it to reach its full IPC potential. With the ability to deploy post-silicon application-specific microarchitectural components, PSM is mostly orthogonal to techniques which target compute acceleration. On the other hand, techniques discussed for configuring the reconfigurable fabric (e.g., using configuration caches to reduce reconfiguration delay [CHW00]) would carry forward directly to our work, and is complementary. We briefly discuss the different styles of integrating reconfigurable logic with the core that have been explored in prior works in the subsections below.

5.1.1 Tightly-integrated reconfigurable functional units

One approach is to tightly integrate reconfigurable logic as customizable functional units close to the execution unit of the core ([YMHB00], [RS94], [GHS11] etc.). Cores leverage the reconfigurable functional units via an expanded instruction set. These approaches typically don’t buffer any state in the reconfigurable fabric and rely on the core to handle the loads and stores from the memory hierarchy.

Designs like PRISC [RS94] and Chimaera [YMHB00] use an FPGA-like fabric and target groups of instructions that can be collapsed into a faster, more complex, operation in the reconfigurable logic, but otherwise act as ‘just another’ functional unit available in the execution stage of the core. DySER [GHS11], on the other hand, utilizes a circuit-switched CGRA-like custom fabric and expands the target to acyclic program sub-paths (encompassing several basic blocks). Regions targeted by DySER are split into a load-slice (that runs on the core) and a computation-slice (that runs on the reconfigurable fabric). The data for the computation-slice is sent to the reconfigurable logic using special instructions. After the

design synthesized in the fabric finishes its computations, it sends the outputs to the core which stores it in the memory, if needed.

5.1.2 Reconfigurable ‘co-processors’

Reconfigurable logic can also be added to the cores such that they act like loosely-coupled co-processors ([CHW00], [GFA⁺11], etc.). These designs include their own communication paths to the memory hierarchy, and can run independently, once started. The core supplies the input data (for the targeted code regions) to the dynamically synthesised designs, and stalls while these co-processors finish their execution.

As an example, [CHW00] describes the GARP architecture which uses an FPGA-style reconfigurable fabric to allow building custom accelerators. Cores offload computation of hot loops to the synthesized design which can communicate with the caches to finish its execution independently. To expand the target of the design to large loops with complex control flow, the GARP compiler takes cues from previous VLIW research to build hyperblocks (with loop-back edges) that can be offloaded to the synthesized co-processor. BERET [GFA⁺11] takes a similar approach where recurring traces (with a high probability of loop-back) are offloaded to a coarse-grained custom fabric for independent execution, until a trace exit is detected. The traces are broken into small data flow subgraphs which were found to be common across multiple workloads. The reconfigurable fabric is built from components which can efficiently execute these segmented subgraph blocks.

5.1.3 Reconfigurable fabric in CMPs

In [WA11], Watkins and Albonesi propose ReMAP which expands the applicability of reconfigurable fabric to multi-threaded applications. They show that a pool of specialized programmable logic (SPL) can be added to a heterogeneous CMP, where the SPL can be partitioned into private chunks for use by the individual cores, or be shared among multiple

cores in the tile to accelerate workloads with multiple threads. ReMAP allows implementation of previously-proposed hardware-based synchronization and communication techniques in the programmable fabric. Applications can be modified (with new instructions) to take advantage of the fast producer-consumer communication and barrier synchronization hardware, instead of communicating/synchronizing through the memory. Compared to previous proposals of dedicated hardware-based synchronization or communication, ReMAP’s reconfigurable fabric offers the advantage that inter-thread communication can be augmented with in-fabric computations to reduce thread load, or better balance them, leading to even better performance. Since ReMAP focuses not only on computation acceleration, but also on better/faster thread synchronization and communication, they are probably closest in spirit to our work.

5.2 Prior work related to PSM Use Cases

Some of the use cases for Post-Silicon Microarchitecture discussed in this work have related works which were proposed as dedicated designs for a general-purpose core. For example, our work adapts Control Flow Decoupling [STR12] for implementation as a PSM design while also increasing its scope of applicability by allowing targeting of control-dependent control flow with the help of additional structures kept in the reconfigurable fabric to fix the branch predictions. Similarly, the prefetching use case can be viewed as related to the Load-Slice Core Microarchitecture [CHA⁺15]. In [CHA⁺15], Carlson et. al. extend a simple in-order core with a bypassing execution path for memory access instructions. Using a novel iterative algorithm, they dynamically determine the backward-slice of the delinquent loads. This slice can use the added execution path to efficiently increase the memory-level parallelism, leading to better performance. This is similar to the prefetching use cases, except that the backward-slices of delinquent loads have been statically determined in our work rather than dynamically learned, although, with the available ports in the PSM

interface, it's quite feasible to do this dynamically in PSM as well. As such, the proposed Post-Silicon Microarchitecture can be viewed as subsuming these microarchitectural proposals by allowing them to be instantiated in the reconfigurable logic, as per an application's needs, instead of adding dedicated silicon for each of the individual designs.

Another set of related works include helper threads or leader-follower style of architectures like Slipstream [SPR00]. Designs like Slipstream can be viewed and adopted as another use case of PSM, where the IR (Instruction Removal) predictor/detector and the delay buffer can be synthesized in the PSM-RF. The overriding branch prediction mechanism provided by the PSM-Agent, along with the ability to snoop the retire stream (to train the IR predictor), can facilitate implementation of Slipstream as a PSM design.

CHAPTER

6

CONCLUSION

This dissertation presented a design and discussion of Post-Silicon Microarchitecture, a new paradigm which fundamentally changes the trade-offs in implementing microarchitectural features and ideas that work well for some, but not all applications. We advocated integrating a reconfigurable fabric on the same chip as a superscalar Out-of-Order core to help alleviate the application-specific microarchitectural bottlenecks. Designs synthesized in the reconfigurable logic intervene transparently in a core’s microarchitectural execution to unlock more performance. We presented a possible design of the interface between the core and the reconfigurable fabric that adheres to the guidelines of simplicity and non-intrusiveness, while being powerful enough to influence the core microarchitecture in meaningful ways.

We highlighted the issue with developing microarchitectural components in the slower

reconfigurable fabric in the presence of a potentially high round-trip latency of communication with the faster core, resulting in an additional requirement of timeliness of predictions. Several use cases for the SPEC benchmarks were showcased that solve the limitation of the slower clock rate of the reconfigurable logic via development of decoupled predictor designs that can run ahead of the core's instruction stream and provide timely predictions to unlock higher performance in the baseline core. With designs of custom branch predictors and prefetchers targeted at varied workloads, we demonstrated the viability of the Post-Silicon Microarchitecture paradigm as a promising road-map to continue the trend of generational improvements in single-thread performance at a time when traditional methods of performance extraction have reduced in efficacy.

REFERENCES

- [AML⁺08] Kubilay Atasu, Oskar Mencer, Wayne Luk, Can Ozturan, and Gunhan Dunder. Fast custom instruction identification by convex subgraph enumeration. In *2008 International Conference on Application-Specific Systems, Architectures and Processors*, pages 1–6. IEEE, 2008.
- [AOFR10] Muawya Al-Otoom, Elliott Forbes, and Eric Rotenberg. EXACT: Explicit dynamic-branch prediction with active updates. In *Proceedings of the 7th ACM international conference on Computing frontiers*, pages 165–176, 2010.
- [AOSR12] Muawya Al-Otoom, Rami Sheikh, and Eric Rotenberg. A case for a software-managed reconfigurable branch predictor. In *5th Workshop on Architectural and Microarchitectural Support for Binary Translation*. Citeseer, 2012.
- [AS93] Peter M. Athanas and Harvey F. Silverman. Processor reconfiguration through instruction-set metamorphosis. *Computer*, 26(3):11–18, 1993.
- [BLvK18] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. SPEC CPU2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 41–42, 2018.
- [CC01] Jorge E Carrillo and Paul Chow. The effect of reconfigurable units in superscalar processors. In *Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, pages 141–150, 2001.
- [CCG⁺15] Yu-Ting Chen, Jason Cong, Michael Gill, Glenn Reinman, and Bingjun Xiao. Customizable computing. *Synthesis Lectures on Computer Architecture*, 10(3):1–118, 2015.
- [CDA03] Lei Chen, Steve Dropsho, and David H Albonesi. Dynamic data dependence tracking and its application to branch prediction. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.*, pages 65–76. IEEE, 2003.
- [CFHZ04] Jason Cong, Yiping Fan, Guoling Han, and Zhiru Zhang. Application-specific instruction generation for configurable processor architectures. In *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 183–189, 2004.
- [CGG⁺14] Jason Cong, Mohammad Ali Ghodrati, Michael Gill, Beayna Grigorian, Karthik Gururaj, and Glenn Reinman. Accelerator-rich architectures: Opportunities and progresses. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2014.
- [CHA⁺15] Trevor E Carlson, Wim Heirman, Osman Allam, Stefanos Kaxiras, and Lieven Eeckhout. The load slice core microarchitecture. In *2015 ACM/IEEE 42nd Annual*

- International Symposium on Computer Architecture (ISCA)*, pages 272–284. IEEE, 2015.
- [Cha19] Aayush Chaudhary. Custom exact branch predictor for astar benchmark. Master’s thesis, North Carolina State University, Raleigh, NC, 2019.
- [CHM⁺14] Jason Cong, Hui Huang, Chiyuan Ma, Bingjun Xiao, and Peipei Zhou. A fully pipelined and dynamically composable architecture of CGRA. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 9–16. IEEE, 2014.
- [CHW00] Timothy J Callahan, John R Hauser, and John Wawrzynek. The Garp architecture and C compiler. *Computer*, 33(4):62–69, 2000.
- [CKP⁺04] Nathan Clark, Manjunath Kudlur, Hyunchul Park, Scott Mahlke, and Krisztian Flautner. Application-specific processing on a general-purpose core via transparent instruction set customization. In *37th international symposium on microarchitecture (MICRO-37’04)*, pages 30–40. IEEE, 2004.
- [CLN⁺11] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011.
- [CM10] Philippe Coussy and Adam Morawiec. *High-level synthesis*, volume 1. Springer, 2010.
- [CST⁺06] Luis Ceze, Karin Strauss, James Tuck, Josep Torrellas, and Jose Renau. CAVA: Using checkpoint-assisted value prediction to hide L2 misses. *ACM Transactions on Architecture and Code Optimization (TACO)*, 3(2):182–208, 2006.
- [FW14] Babak Falsafi and Thomas F Wenisch. A primer on hardware prefetching. *Synthesis Lectures on Computer Architecture*, 9(1):1–67, 2014.
- [GC07] Philip Garcia and Katherine Compton. A reconfigurable hardware interface for a modern computing system. In *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007)*, pages 73–84. IEEE, 2007.
- [GFA⁺11] Shantanu Gupta, Shuguang Feng, Amin Ansari, Scott Mahlke, and David August. Bundled execution of recurring traces for energy-efficient general purpose processing. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 12–23, 2011.
- [GHS11] Venkatraman Govindaraju, Chen-Han Ho, and Karthikeyan Sankaralingam. Dynamically specialized datapaths for energy efficient computing. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 503–514. IEEE, 2011.

- [GJN11] Marius Grannaes, Magnus Jahre, and Lasse Natvig. Storage efficient hardware prefetching using delta-correlating prediction tables. *Journal of Instruction-Level Parallelism*, 13:1–16, 2011.
- [GLM10] Antonio Gonzalez, Fernando Latorre, and Grigorios Magklis. Processor microarchitecture: An implementation perspective. *Synthesis Lectures on Computer Architecture*, 5(1):1–116, 2010.
- [GMDZ08] Hongliang Gao, Yi Ma, Martin Dimitrov, and Huiyang Zhou. Address-branch correlation: A novel locality for long-latency hard-to-predict branches. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pages 74–85. IEEE, 2008.
- [GSM⁺99] Seth Copen Goldstein, Herman Schmit, Matthew Moe, Mihai Budiu, Srihari Cadambi, R Reed Taylor, and Ronald Laufer. PipeRench: A coprocessor for streaming multimedia acceleration. In *Proceedings of the 26th International Symposium on Computer Architecture (Cat. No. 99CB36367)*, pages 28–39. IEEE, 1999.
- [GT07] Brian Greskamp and Josep Torrellas. Paceline: Improving single-thread performance in nanoscale CMPs through core overclocking. In *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, pages 213–224. IEEE, 2007.
- [Hen06] John L Henning. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.
- [Jai19] Saransh Jain. Application specific parallelization. Master’s thesis, North Carolina State University, Raleigh, NC, 2019.
- [KCKY01] Nicholas Kohout, Seungryul Choi, Dongkeun Kim, and Donald Yeung. Multi-chain prefetching: Effective exploitation of inter-chain memory parallelism for pointer-chasing codes. In *Proceedings 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 268–279. IEEE, 2001.
- [KGV04] Georgi Kuzmanov, Georgi Gaydadjiev, and Stamatis Vassiliadis. The MOLEN processor prototype. In *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 296–299. IEEE, 2004.
- [KT99] Venkata Krishnan and Josep Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers*, 48(9):866–880, 1999.
- [KTR08] Ian Kuon, Russell Tessier, and Jonathan Rose. Fpga architecture: Survey and challenges. *Foundations and Trends® in Electronic Design Automation*, 2(2):135–253, 2008.

- [LA18] Ankur Limaye and Tosiron Adegbiya. A workload characterization of the SPEC CPU2017 benchmark suite. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 149–158. IEEE, 2018.
- [LBR09] Marco Lanzagorta, Stephen Bique, and Robert Rosenberg. Introduction to reconfigurable supercomputing. *Synthesis Lectures on Computer Architecture*, 4(1):1–103, 2009.
- [LJLS08] Jaejin Lee, Changhee Jung, Daeseob Lim, and Yan Solihin. Prefetching with helper threads for loosely coupled multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 20(9):1309–1324, 2008.
- [Mat19] Utkarsh Mathur. Post-silicon microarchitecture (psm) implementation of checkpointed early load retirement (clear). Master’s thesis, North Carolina State University, Raleigh, NC, 2019.
- [MD⁺96] Ethan Mirsky, Andre DeHon, et al. MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources. In *FCCM*, volume 96, pages 17–19, 1996.
- [MF93] Scott Mc Farling. WRL Technical Note TN-36. *Combining Branch Predictors*. Jun, 1993.
- [MPJ⁺00] Ken Mai, Tim Paaske, Nuwan Jayasena, Ron Ho, William J Dally, and Mark Horowitz. Smart memories: A modular reconfigurable architecture. In *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No. RS00201)*, pages 161–171. IEEE, 2000.
- [MSK⁺99] Alan Marshall, Tony Stansfield, Igor Kostarnov, Jean Vuillemin, and Brad Hutchings. A reconfigurable arithmetic array for multimedia applications. In *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pages 135–143, 1999.
- [MSWP03] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.*, pages 129–140. IEEE, 2003.
- [MVV⁺03] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. *IEE Proceedings-Computers and Digital Techniques*, 150(5):255–261, 2003.
- [NGS15] Tony Nowatzki, Vinay Gangadhar, and Karthikeyan Sankaralingam. Exploring the potential of heterogeneous von neumann/dataflow execution models. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 298–310, 2015.

- [PCC⁺14] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale data-center services. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 13–24. IEEE, 2014.
- [PPM09] Hyunchul Park, Yongjun Park, and Scott Mahlke. Polymorphic pipeline array: a flexible multicore accelerator with virtualized execution for mobile multimedia applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 370–380, 2009.
- [PSR00] Zach Purser, Karthik Sundaramoorthy, and Eric Rotenberg. A study of slipstream processors. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 269–280, 2000.
- [RAJ00] Parthasarathy Ranganathan, Sarita Adve, and Norman P Jouppi. Reconfigurable caches and their application to media processing. *ACM SIGARCH Computer Architecture News*, 28(2):214–224, 2000.
- [RLG⁺98] Charle R Rupp, Mark Landguth, Tim Garverick, Edson Gomersall, Harry Holt, Jeffrey M Arnold, and Maya Gokhale. The NAPA adaptive processing architecture. In *Proceedings. IEEE Symposium on FPGAs for Custom Computing Machines (Cat. No. 98TB100251)*, pages 28–37. IEEE, 1998.
- [Rot18] E Rotenberg. FoMR: Post silicon micro-architecture, NSF grant no. CCF-1823517. 2018.
- [RS94] Rahul Razdan and Michael D Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Proceedings of the 27th annual international symposium on Microarchitecture*, pages 172–180, 1994.
- [Sez16] André Seznec. TAGE-sc-l branch predictors again. In *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, 2016.
- [SKB⁺15] Manjunath Shevgoor, Sahil Koladiya, Rajeev Balasubramonian, Chris Wilkerson, Seth H Pugsley, and Zeshan Chishti. Efficiently prefetching complex address patterns. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 141–152. IEEE, 2015.
- [SPHC02] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. *ACM SIGPLAN Notices*, 37(10):45–57, 2002.
- [SPR00] Karthik Sundaramoorthy, Zach Purser, and Eric Rotenberg. Slipstream processors: Improving both performance and fault tolerance. *ACM SIGPLAN Notices*, 35(11):257–268, 2000.

- [Sri19] Vinesh Srinivasan. *Slipstream Processors Revisited: Exploiting Branch Sets*. PhD thesis, North Carolina State University, Raleigh, NC, 2019.
- [STR12] Rami Sheikh, James Tuck, and Eric Rotenberg. Control-flow decoupling. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 329–340. IEEE, 2012.
- [TK12] Vaishali Tehre and Ravindra Kshirsagar. Survey on coarse grained reconfigurable architectures. *International Journal of Computer Applications*, 48(16):1–7, 2012.
- [VSG⁺10] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation cores: reducing the energy of mature computations. *ACM Sigplan Notices*, 45(3):205–218, 2010.
- [WA11] Matthew A Watkins and David H Albonesi. ReMAP: A reconfigurable architecture for chip multiprocessors. *IEEE micro*, 31(1):65–77, 2011.
- [WC95] Ralph D Wittig and Paul Chow. OneChip: An FPGA Processor With Reconfigurable Logic. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 126–135, 1995.
- [WCA08] Matthew A Watkins, Mark J Cianchetti, and David H Albonesi. Shared reconfigurable architectures for CMPS. In *2008 International Conference on Field Programmable Logic and Applications*, pages 299–304. IEEE, 2008.
- [WH95] Michael J Wirthlin and Brad L Hutchings. A dynamic instruction set computer. In *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*, pages 99–107. IEEE, 1995.
- [WLP⁺14] Andrew Waterman, Yunsup Lee, David Patterson, Krste Asanovic, and Volume I User level Isa. The RISC-V instruction set manual. *Volume I: User-Level ISA, version, 2*, 2014.
- [YM04] Pan Yu and Tulika Mitra. Scalable custom instructions identification for instruction-set extensible processors. In *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 69–78, 2004.
- [YMHB00] Zhi Alex Ye, Andreas Moshovos, Scott Hauck, and Prithviraj Banerjee. CHI-MAERA: a high-performance architecture with a tightly-coupled reconfigurable functional unit. *ACM SIGARCH Computer Architecture News*, 28(2):225–235, 2000.
- [YRHK06] Dong Ye, Joydeep Ray, Christophe Harle, and David Kaeli. Performance characterization of SPEC CPU2006 integer benchmarks on x86-64 architecture. In *2006 IEEE International Symposium on Workload Characterization*, pages 120–127. IEEE, 2006.

- [ZQLT12] Fucen Zeng, Lin Qiao, Mingliang Liu, and Zhizhong Tang. Memory performance characterization of spec cpu2006 benchmarks using tsim. *Physics Procedia*, 33:1029–1035, 2012.