ABSTRACT

SESHADRI, ANIRUDH. Delinquent Loop Pre-execution Using Predicated Helper Threads (Phelps). (Under the direction of Dr. Eric Rotenberg).

Data-dependent conditional branches are infamous for their incompatibility with history-based branch predictors resulting in poor prediction accuracy. These branches could result in frequent branch mispredictions when they are found to be nested within loops. Thus, contemporary processors, with aggressive front ends, could be fetching down the incorrect instruction stream for many cycles limiting the processor's ability to build up a useful instruction window and also wasting energy due to executing instructions down the wrong path. Branch pre-execution is a potential solution for handling such hard-to-predict (delinquent) branches. It does so by pre-executing the computation leading up to the delinquent branches ahead of time using redundant helper threads. However, prior work in branch pre-execution may result in low accuracy and/or late pre-execution when (1) a delinquent branch b2 is control-dependent on another delinquent branch b1 and/or (2) a store instruction both influences a delinquent branch and is control-dependent on one. My research focuses on addressing these issues. I present my two primary contributions to the design space of branch pre-execution.

My first contribution in this space is to deploy custom branch predictors in a Post-Fabrication Microarchitecture paradigm. I showcase my custom branch predictor designs for the *astar* benchmark from SPEC 2006 and the *bfs* benchmark from the GAP benchmark suite. Both custom predictors essentially pre-execute delinquent branches in the region of interest, by mimicking the computation leading to these branches through a customized datapath. Nested delinquent branches b1 and b2 are unconditionally pre-executed, excess predictions of b2 are correctly filtered out, and the effects of stores are inferred.

My second contribution in the branch pre-execution design space is Predicated Helper Threads (Phelps). Phelps is a fully-automated hardware-only branch pre-execution paradigm

that focuses on pre-executing loops containing delinquent branches. Phelps dynamically detects delinquent branches and the loops that contain them, and constructs a helper thread for each such loop. All delinquent branches in the loop, such as nested delinquent branches b1 and b2, are unconditionally pre-executed. Per-branch prediction queues are managed in lock-step based on loop iterations. This enables (1) the helper thread to deposit outcomes of b1 and b2 for each loop iteration, and (2) the main thread to use or not use outcomes of b2 according to b1's outcomes. Overall, this approach ensures that the pre-execution of nested delinquent branches is both timely and accurate. A helper thread also retains influential stores for dynamic disambiguation and store-load forwarding (without affecting architectural state of the main thread), and any such store that is control-dependent on a delinquent branch is predicated on that branch's outcome. A novel control-dependency (CD) analysis technique, based on simple hardware FSMs (that render reconvergent point detection unnecessary), is used to identify the branch that guards the store. Phelps also has the ability to create dual decoupled helper threads for outer-inner loop pairs to increase memory level parallelism.

Delinquent Loop Pre-execution Using Predicated Helper Threads (Phelps)

by
Anirudh Seshadri

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
2024

APPROVED BY:

Dr. Huiyang Zhou	Dr. Samira Mirbagher Ajorpaz
Dr. Xipeng Shen	Dr. Eric Rotenberg Chair of Advisory Committee

DEDICATION

To my grandmother whom I couldn't kiss a goodbye.

BIOGRAPHY

Anirudh was born in Chennai, India on 16th June 1995. He earned his Bachelors of Technology in Information Technology from Anna University, Chennai, India in Jan 2018. Anirudh was a student research trainee at Waran Research Foundation (WARFT,India) before joining NC State for pursuing a Master of Science in Computer Engineering degree in Fall 2018. After joining NC State, Anirudh started working as a graduate research assistant under the supervision of Dr. Eric Rotenberg from Jan 2020. Highly motivated and interested in the research, Anirudh switched over from the masters program to pursue a Ph.D. degree in Computer Engineering from Jan 2020.

ACKNOWLEDGEMENTS

I would like to take this space to express my sincere gratitude to everyone who has been a part of my journey here at NCSU. First and foremost, I would like to thank Dr. Eric Rotenberg for his continued guidance and patience in nudging me in the right direction throughout my PhD. As someone entirely new to the field of microarchitecture, his captivating lectures inspired me to pursue research in Microarchitecture. His immense knowledge in the field and his experience in research have helped shape my project and my career profile. His relentless drive for excellence and perfection are qualities I wish to develop and carry with me throughout my life.

I would also like to thank the members of my PhD committee for their cooperation, valuable insights, and feedback. I thank the amazing teaching faculty in the department of ECE at NCSU for their well-structured coursework that really sets students up for their career.

I am extremely grateful to have had Jiayang Li, Rohit Singh, and Zach Johnston as my lab mates during my time at NCSU. Their experience and knowledge in the simulator infrastructure has helped me immensely. I have also learned a lot just from their conversations with my advisor during our group meetings. A huge thanks to Chanchal Kumar for providing me with the opportunity to be a part of the Post-Fabrication Microarchitecture project. Special thanks to Shubham Bhawalkar, for always being available for a discussion or a fun chat or sometimes both. I am extremely grateful for having him as a friend that I can approach for all things technical and non-technical. I would also like to extend my sincere thanks to Prof. Venkateswaran Natarajan for his guidance during my undergraduate studies.

Words cannot express the gratitude I have for my wife, Gayathri, for the sacrifices she has made and for putting up with me during this phase of life which had several ups and downs. Likewise, the constant support and backing I have from my parents and my sister have really helped me immensely. Thanks to my grandparents, brother-in-law, my uncles,

aunts, and cousins for their backing and believing in me. My cute little nephew also deserves a shout out for always keeping my spirits high.

My friends and roommates over the years – Hari, Suriya, Arul, Prabhu, Varun, Raghul, Prashanth, Naveen, Arpitha, and Abhijith – are very much appreciated for making me feel home even though I was far away from home.

This research was funded by the NSF/Intel Partnership on Foundational Microarchitecture Research (FoMR) (NSF grant no. CCF-1823517 and matching Intel grant). 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 Corporation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Data-dependent conditional branches are infamous for their incompatibility with history-based branch predictors, resulting in poor prediction accuracy. These branches could result in frequent branch mispredictions when they are found to be nested within loops. Thus, contemporary processors, with aggressive front ends, could be fetching down the incorrect instruction stream for many cycles limiting the processor's ability to build up a useful instruction window and also wasting energy due to executing instructions down the wrong path. Such frequently mispredicted hard-to-predict branches are often termed delinquent branches and can be typically observed in graph-based workloads. Graph workloads tend to iterate through a worklist to pop a node, test the node's property, may optionally even modify its property, and add additional candidate nodes to the worklist. In this case, the contents of the worklist are highly dynamic and this causes the branch that tests the property of the node to not show any correlation with prior instances of this or other branches. Such branches would not benefit from history-based branch predictors.

One way to overcome such hard-to-predict branches is through a technique called branch predication (*e.g.*, [AKPW83; MHM+95; KAGC98; CGS+20]). Branch predication removes the delinquent branch from the instruction stream, fetches and issues instructions down both its paths, but conditionally commits instructions that are in the correct path. This is

achieved by converting control dependencies on the delinquent branch into data dependencies on the delinquent branch's predicate. By removing the branches from the instruction stream, predication benefits through a more streamlined instruction fetch stream by virtue of fewer mispredictions and fewer discontinuities. However, this benefit comes at the cost of executing more instructions and the cost outweighs the benefits as soon as we encounter large CD regions. Furthermore, it may be difficult to predicate complex CD regions that contain nested conditional branches, function calls, loops, etc.

Branch pre-execution is another well-known technique to handle delinquent conditional branches (*e.g.*, [CSK+99; PSR00; RS01; ZS01; CTYP02; GH08; PH14; KH19; SCR20; KSC+21; PP21]). It does so by pre-executing the computation leading up to the delinquent branch ahead of time using redundant helper thread(s). The forked helper thread(s) comprises the instructions that are in the backward slice of the delinquent branch and are intended to run ahead of the main thread's fetch stream and evaluate the outcome of the delinquent branch. These outcomes are then communicated back to the main thread via communication queues resulting in high branch prediction accuracy. The accuracy (correctness) of the pre-executed outcomes as well as the timeliness of pre-execution are vital to ensure that the benefit of pre-execution outweighs the cost incurred in redundantly executing the helper thread instructions.

My research is motivated by the need to address limitations of the prior art in branch pre-execution. In particular, **prior approaches may result in low accuracy and/or late pre-execution if the backward slice of the targeted delinquent branch contains (1) *other delinquent branches* and/or (2) *stores***. Figures 1.1, 1.2, and 1.3 illustrate the issues. The column (a) in each of these figures depicts a pseudo-CFG of a code snippet with *only* the instructions in the backward slice of the targeted delinquent branch being highlighted. Also, note that there may be other instructions in the backward slice before the first shown instruction, but these are not shown since they are not relevant to the following discussion.

Figure 1.1 illustrates the case where the delinquent branch is control dependent (CD) on another branch ("guard" branch). The delinquent branch i20 is dependent on the guard branch i10's predicate evaluating as not taken (NT) for its existence. Thus, it is imperative to handle such branches to correctly correlate pre-execution outcomes from the helper thread with the main thread's fetch stream. When the guard branch i10 is highly biased (as in Figure 1.1b), including just the instructions from the biased path and pruning i10 from the backward slice would still lead to a correlated pre-execution stream. When i10 is not biased but highly predictable, simply including i10 in the backward slice, and predicting its direction during pre-execution could assist the helper thread with the decision of generating an outcome for i20. This is shown in Figure 1.1c.

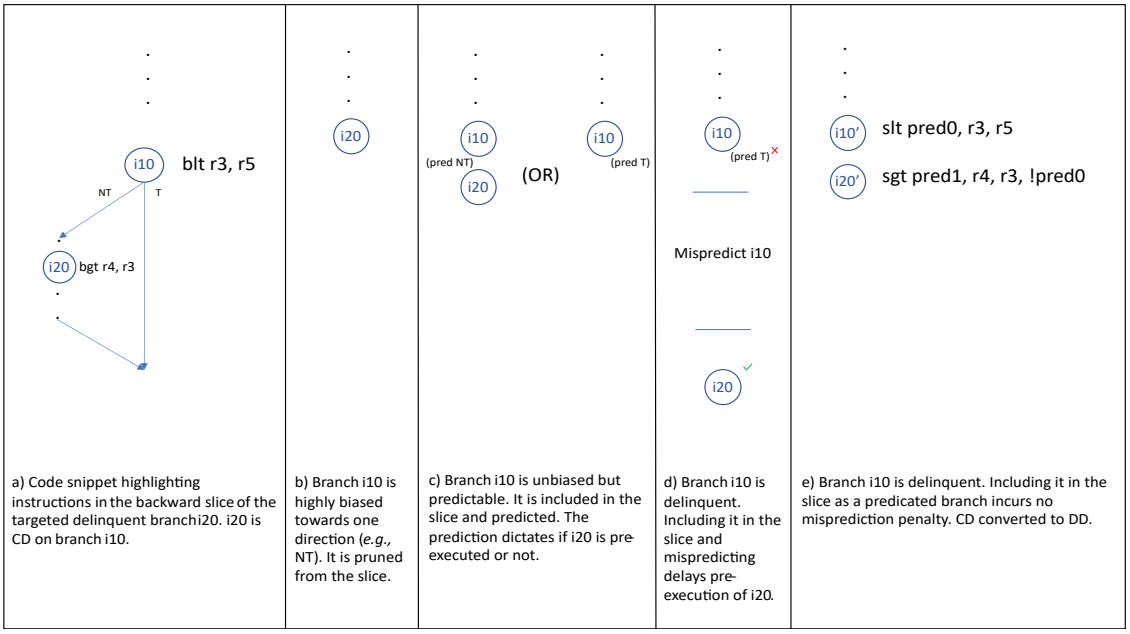


Figure 1.1: Targeted delinquent branch CD on another branch.

The issue is again when the guard branch is unbiased, unpredictable, and thus *itself delinquent* as seen in Figure 1.1d. Branch i10 cannot be pruned from the backward slice as it is unbiased. Including i10 in the slice and predicting its direction could prove ineffective due to delayed pre-execution. This is because, owing to the unpredictability of i10, its frequent

mispredictions might have to retrigger the pre-execution of i20 after incurring a long misprediction penalty. This delayed pre-execution again affects the accuracy of pre-execution if the default predictor is used when pre-execution is not timely. This makes a case for eagerly pre-executing the nested delinquent branch i20 within the helper thread irrespective of the outcome of i10. However, doing so requires the ability to ignore/discard prediction outcomes that were supposed to be skipped over due to i10 resolving as taken. One possible way to achieve this is using lockstep-operated per-PC prediction queues as we would see in Chapter 5.

The second issue existing pre-execution techniques fail to handle efficiently is when the backward slice of the delinquent branch involves data flowing through memory. This happens when a load instruction in the backward slice reads an address in memory that was recently written to by a store in the same loop as the delinquent branch as seen in Figure 1.2.

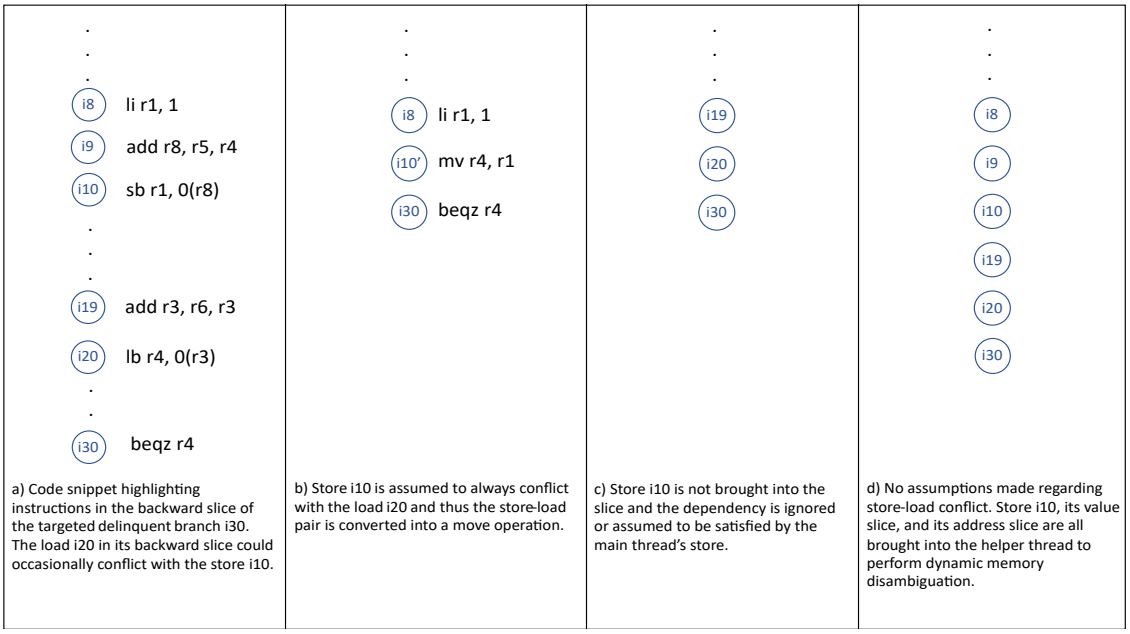


Figure 1.2: Store is in the backward slice of the targeted delinquent branch.

Figure 1.2a shows the targeted delinquent branch i30 being dependent on the value loaded by i20. Furthermore, load i20 may occasionally conflict with the store i10. Different choices

may be made to handle such memory dependencies. For instance, it may be assumed that the static store and the static load always conflict and thus, they can be converted into a move operation in the helper thread as shown in Figure 1.2b. This assumption could lead to lots of mispredictions if the nature of the store-load conflict is highly dynamic. Another policy is to not include the store, and its backward slice in the helper thread, as shown in Figure 1.2c. Such a policy would yield correct pre-execution outcomes when either (1) store i10 and load i20 do not conflict, or (2) store i10 is completed by the main thread before its conflicting load i20 issues in the helper thread. When both the above conditions are not met, i20 loads a stale value which could lead to incorrect pre-execution of the delinquent branch i30. These issues make the case for including the store i10, its value slice, and its address slice in the helper thread and performing dynamic memory disambiguation in the helper thread as it executes (shown in Figure 1.2d).

The issue with stores is further exacerbated when the store is guarded by another delinquent branch. Figure 1.3a shows one such case where the store i10, which is in the backward slice of the targeted delinquent branch i30, is control dependent on another delinquent branch i7. This deems i30 to be CIDD on i7. Including i7 in the backward slice and predicting it is ineffective as it is delinquent. The resultant misprediction ensues a long delay in pre-executing i30. Thus, converting i7 into a predicated branch might be the most effective way to pre-execute i30. This is shown in Figure 1.3b where the store i10 is predicated on the outcome of the delinquent branch i7.

Contribution 1: My first contribution in this space is to deploy custom branch predictors in the Post-Fabrication Microarchitecture paradigm [KSC+21]. I showcase my custom branch predictor designs for the *astar* benchmark from SPEC 2006 and the *bfs* benchmark from the GAP benchmark suite. Both custom predictors essentially pre-execute delinquent branches in the region of interest, by mimicking the computation leading to these branches through a customized datapath. All delinquent branches are unconditionally pre-executed, including the control-dependent ones. Leveraging insights from the region of interest, the

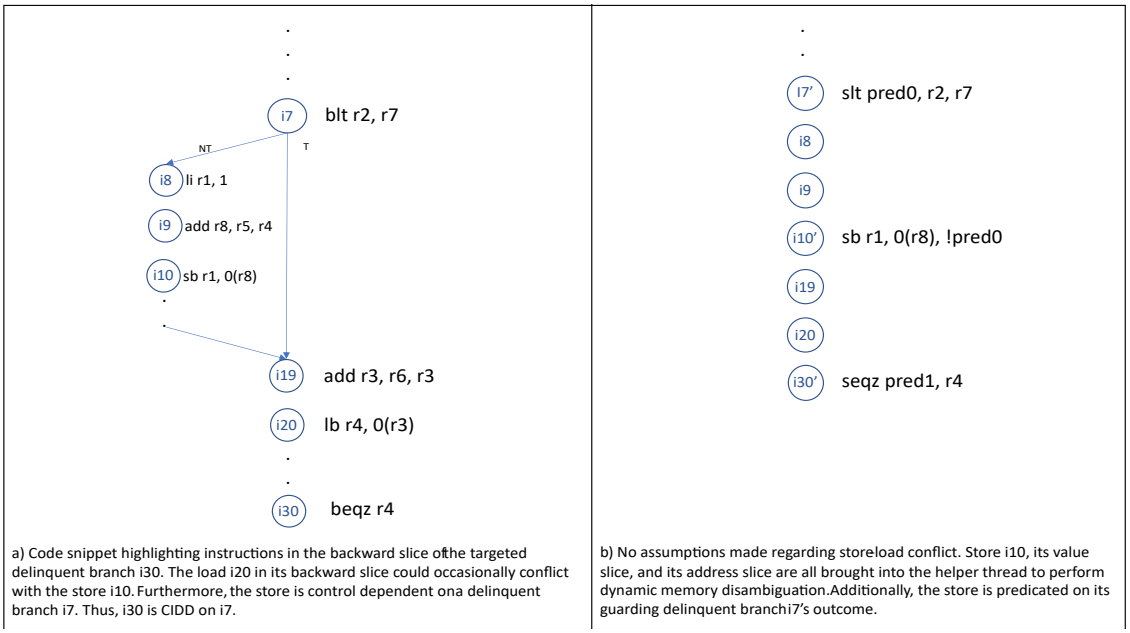


Figure 1.3: Store is in the backward slice of the targeted delinquent branch. The store is CD on another delinquent branch.

custom predictor correctly filters out excess predictions of control-dependent branches and infers the effects of stores.

Contribution 2: My second contribution in the branch pre-execution design space is Predicated Helper Threads (Phelps). Phelps is a fully-automated hardware-only branch pre-execution paradigm that focuses on pre-executing loops containing delinquent branches. Phelps dynamically detects delinquent branches and the loops that contain them, and constructs a helper thread for each such loop. All delinquent branches in the loop, including control-dependent ones, are unconditionally pre-executed. Per-branch prediction queues are managed in lock-step based on loop iterations. This enables (1) the helper thread to deposit outcomes of all delinquent branches for each loop iteration, and (2) the main thread to use or not use outcomes of a control-dependent branch based on the outcome of its guarding branch. Overall, this approach ensures that the pre-execution of nested delinquent branches is both timely and accurate. A helper thread also retains influential stores for dynamic disambiguation and store-load forwarding (without affecting architectural state of the main

thread), and any such store that is control-dependent on a delinquent branch is predicated on that branch's outcome. A novel control-dependency (CD) analysis technique, based on simple hardware FSMs (that render reconvergent point detection unnecessary), is used to identify the branch that guards the store.

Phelps also constructs dual decoupled helper threads in the case of outer-inner loop pairs that contain delinquent branches. This decoupling enables the outer thread to run ahead and queue multiple visits to the inner loop for the inner thread to process. The outer thread's progress is unaffected by the inner thread, generating misses far ahead into the instruction stream, thereby increasing memory level parallelism (MLP). Leveraging MLP due to decoupling of nested loops, along with Phelps' handling of nested delinquent branches and stores, enables Phelps to achieve timely and accurate pre-execution of the delinquent branches.

This document is organized as follows. Chapter 2 focuses on the related work in the field of branch pre-execution. Chapter 3 motivates predicated helper threads by closely examining the state-of-art branch pre-execution technique, Branch Runahead [PP21], and highlighting its challenges. Chapter 4 delves deeper into the first contribution of my research, custom pre-execution engines deployed as post-fabrication microarchitecture. Chapter 5 describes my second contribution, Phelps. Chapter 6 presents performance and analysis of Phelps on various benchmarks. Finally, Chapter 7 summarizes my research contributions.

Chapter 2

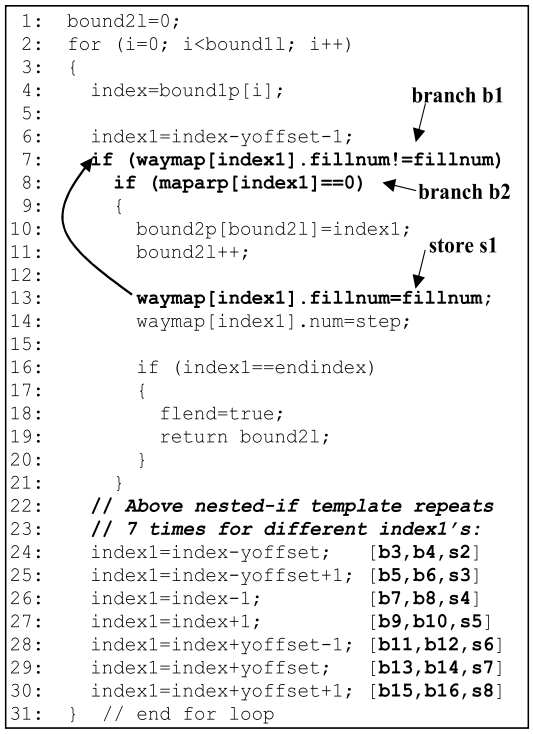
Related Work

Branch pre-execution is a well-explored area of research that saw a flurry of activity in the 2000s. It has been recently revisited [SCR20; PP21; KSC+21]. This chapter summarizes all of this prior work using a sample region of interest from the SPEC 2006 *astar* benchmark to highlight their limitations.

2.1 Astar ROI

I will use a sample region of interest (ROI) to walk through the state-of-the-art branch pre-execution schemes and highlight their limitations. The code snippet in Figure 2.1 corresponds to the body of the most prominent functions that appear in the top-weighted simpoint [SPHC02] of the SPEC benchmark, astar (for the rivers dataset). astar is a search algorithm used to find the optimal path between two nodes in a graph and as most graph applications do, astar has a lot of data-dependent branches that are naturally hard to predict due to their poor correlation with prior branch history. The top-weighted simpoint incurs about 29 mispredictions-per-kilo-instructions (MPKI) using a state-of-art 64KB TAGE-SC-L [Sez16] most of which originate from within wayobj::makebound2().

The wayobj::fill() kickstarts a series of visits to nodes in the graph. It does so by repeatedly calling wayobj::makebound2() to iterate through an input worklist of nodes. Each iteration




```
 1:  bound2l=0;
 2:  for (i=0; i<bound1l; i++)
 3:  {
 4:    index=bound1p[i];
 5:
 6:    index1=index-yoffset-1;
 7:    if (waymap[index1].fillnum!=fillnum)
 8:      if (maparp[index1]==0)
 9:      {
10:        bound2p[bound2l]=index1;
11:        bound2l++;
12:
13:        waymap[index1].fillnum=fillnum;
14:        waymap[index1].num=step;
15:
16:        if (index1==endindex)
17:        {
18:          flend=true;
19:          return bound2l;
20:        }
21:      }
22:    // Above nested-if template repeats
23:    // 7 times for different index1's:
24:    index1=index-yoffset;   [b3,b4,s2]
25:    index1=index-yoffset+1; [b5,b6,s3]
26:    index1=index-1;         [b7,b8,s4]
27:    index1=index+1;         [b9,b10,s5]
28:    index1=index+yoffset-1; [b11,b12,s6]
29:    index1=index+yoffset;   [b13,b14,s7]
30:    index1=index+yoffset+1; [b15,b16,s8]
31:  }  // end for loop
```

Figure 2.1: Astar code fragment.

in the loop pops an index from the worklist and generates the indices of the eight nodes, referred to as index1 in the code snippet, that surround the popped node. For each index1 node, two conditional branches namely *waymap* (b1) and *maparp* (b2) test the properties of the node and determine if index1 needs to be explored in the next call to makebound2(). The waymap branch checks if the node has already been visited in the current fill() context. Only nodes that have not yet been visited in the current fill() context *and* satisfy the maparp condition are to be added to the output worklist. The same nested-if template repeats for the seven other index1 generated thereby generating 8 pairs of nested if branches *i.e.*, {b1, b2}, {b3, b4}, ... {b15, b16}. Furthermore, for each of these nested-if regions, a store to the fillnum structure (for e.g., s1) ensures that subsequent visits to the same node are avoided. Thus there is a loop-carried memory dependency between the stores (s1, s2, ... s8) and the waymap branches (b1, b3, b5, ... b15).

Both the waymap and maparp branches are highly mispredicted with history-based

branch predictors. This can be attributed to the highly dynamic nature of the contents of the input worklist (thereby highly dynamic index1 values) and also the loop-carried memory dependence between the stores and the waymap branches. Such branches are good candidates for branch pre-execution, however, state-of-the-art branch pre-execution techniques face key challenges in handling such regions as explained below.

There are two major styles of branch pre-execution. The first style uses a single, continuous, self-sustained helper thread that, once forked, can pre-execute multiple dynamic instances of the delinquent branch [PSR00; ZS01; GH08; PH14; KH19; SCR20; KSC+21]. The second style repeatedly forks multiple, short-lived helper threads, each of which pre-executes a single dynamic instance of the delinquent branch [CSK+99; RS01; CTYP02; PP21]. The former benefits from creating a single continuous helper thread that does not rely on accurately timing the forking of helper threads. The latter benefits from simple helper threads that might not include control-flow instructions but have to carefully time the forking of a helper thread to generate pre-executed outcomes ahead of time and also synchronize with the main thread every time it triggers.

2.2 Prior branch pre-execution work

Chappell et al. [CSK+99] proposed SSMT to utilize the idle thread contexts in an SMT core to benefit the main thread. The idle contexts are used to run microthread routines that can implement complete branch predictors, to serve as an alternative to the core's hardware branch predictor. The authors showcase an example of implementing the PAg predictor in microcode whose predictions could benefit branches that are self-correlated. Despite using helper threads to target branches, SSMT cannot be classified as branch pre-execution since it does not extract the backward slices of delinquent branches to create the helper threads.

In Slipstream (SS) processors, **Purser et al. [PSR00]** propose running two threads in a leader-follower arrangement to attain improved single-thread performance as well as fault tol-

erance. The leader thread is a reduced version (A-stream) of the follower thread (R-stream) by pruning ineffectual instructions and their backward slices from it. Ineffectual instructions include unreferenced writes, non-modifying writes, highly predictable branches, etc. Removing highly predictable branches from the A-stream leaves behind delinquent branches and computation leading to them. The shorter A-stream runs ahead and communicates branch outcomes, and operand values back to the R-stream, which benefits through a streamlined execution. For the makebound2 loop discussed above, Slipstream would retain the waymap branches as well as the maparp branches in the A-stream and also account for the loop-carried memory dependency by retaining the store in the A-stream. However, Slipstream would not find many opportunities to prune instructions from the loop, thereby leading to an A-stream that is almost as big as the R-stream. Furthermore, by predicting the waymap and maparp branches in the A-stream, Slipstream would incur significant misprediction penalties and thus leading to ineffective pre-execution.

Roth and Sohi [**RS01**] manually create data-driven threads that consist of critical computations that lead to problem-causing instructions such as delinquent branches and cache-missed loads. Initiating these threads early enables the main thread to better tolerate the latency due to misses and avoid some branch mispredictions. In addition to pre-executing these instructions, the speculative results produced by the data-driven threads may also be reused by the main thread through register integration. The helper threads created do not contain any control flow and does not provide a mechanism for handling nested delinquent branches.

Zilles and Sohi [**ZS01**] proposed to use speculative slices for pre-executing delinquent branches. The slices are again manually constructed from the backward slice of the delinquent branches that were identified through profiling. They also propose a mechanism for handling control-flow that determines the existence of the delinquent branch, albeit by not including the control-flow in the speculative slice. For instance, in the case of a branch guarding a delinquent branch, only the delinquent branch and its backward slice are included in the

slice. Instead of including the guard branch as part of the slice, the branch outcomes of the delinquent branch are unconditionally pushed into the global prediction queue. They propose a mechanism for deallocating a prediction if it was pushed incorrectly into the queue. For each static branch that is being pre-executed, a set of PCs are manually marked as kill PCs which essentially determines if the branch instance is no longer reachable. When the main thread's control flow reaches one of these kill PCs, their corresponding pre-execution instance is dequeued from the prediction queue. Though not covered in the paper, it can be conceived that when the guard branch is itself delinquent, there can be two independent slices that are concurrently pre-executing the guarding and the guarded delinquent branch in separate thread contexts. Thus, b1 and b2 may be pre-executed by this technique. However, the slices and kill PCs are manually identified and the slices would not contain the store s1 too.

DPSSMT, proposed by **Chappell et al. [CTYP02]**, improves upon their previous work by introducing a mechanism for constructing the helper threads in hardware. Further, they incorporate path history in determining delinquent branches. A path here is defined as the sequence of most recent control flow changes (taken branches). Using path history to determine if a branch is delinquent helps DPSSMT to skip pre-executing instances of delinquent branches that are predictable in some paths. This mechanism is in fact a way to account for control-flow in the backward slice of the branch, as different slices are created for the same delinquent branch based on the path history. However, in cases like the makebound2 loop where the branches (b1, b2, b3, ...) are all delinquent regardless of the path taken, each delinquent branch could end up creating multiple redundant helper threads leading to an explosion of slices.

Garg and Huang [GH08] propose the performance-correctness explicitly-decoupled architecture (DLA). Its performance core runs a well-reduced skeleton of the program to communicate accurate branch predictions and generate prefetching for the full program that is run on the correctness core. The skeleton is obtained through offline profiling of the program's binary. This process eliminates useless branches such as highly biased branches, empty

if-else blocks and empty loops. All other branches in the main program are included in the skeleton and would communicate their outcomes back to the correctness core. Furthermore, only a fraction of stores are included in the skeleton based on the distance between stores and their consumers obtained by profiling. Stores with considerably larger distances are not included in the skeleton with the assumption that the dependency is satisfied by the main program's store. While DLA does have the ability of including the store s1 in the skeleton and also handling nested delinquent branches in the skeleton, its pre-execution efficiency is affected due to the lack of opportunities to prune instructions in the skeleton program.

Parihar and Huang [PH14] improve upon DLA by employing genetic algorithms to identify additional opportunities for pruning the skeleton. Instructions which are involved in weak dependency relationships, such as silent loads, dynamic NOPs, etc., are considered for pruning. **Kondugli's and Huang's R3DLA [KH19]** increases the efficacy of DLA by further *reducing* the skeleton program, *reusing* values produced by instructions in the performance core as a form of value prediction for corresponding instructions in the correctness core (as was done in Slipstream [PSR00]), and *recycles* by dynamically detecting the best skeleton by cycling through candidate skeletons that were each constructed using different combinations of optimizations. These optimizations however, might not still be sufficient to prune instructions in regions that contain mostly delinquent branches.

Srinivasan et al. [SCR20] modify the construction of A-stream in Slipstream2.0 (SS2) to address the ineffectiveness of Slipstream, in accelerating regions/phases with lots of delinquent branches. This is because as explained earlier, in such phases, Slipstream's A-stream is not really shorter compared to its R-stream. Slipstream 2.0 (SS2) creates a single continuous helper thread (referred to as A-stream) to pre-execute the delinquent branches. The A-stream and the R-stream (main thread) are arranged in a leader-follower fashion wherein the A-stream pre-executes the delinquent branches and the R-stream consumes the pre-executed outcomes to achieve good branch prediction accuracy. The A-stream is a shorter version of the R-stream wherein the forward control-flow slice of the delinquent branch

is removed. These include instructions that are control-dependent (CD) on the delinquent branch, and also instructions in the CD region of branches that are control-independent but data-dependent (CIDD) on the delinquent branch.

SS2 uses reconvergent point identification to infer the CD region of a delinquent branch. It then uses the delinquent branch PC, its reconvergent point PC, and a forward poisoning scheme to identify branches that are CIDD on the delinquent branch. Essentially, instructions that are within the CD region of the delinquent branch poisons their destination registers. Instructions beyond the reconvergent point PC, propagate the poison bits from their source registers to their destination register. Branches beyond the reconvergent point PC, that source a poisoned register are CIDD on the delinquent branch. Thus, when the A-stream fetches a delinquent branch or its CIDD branch, the fetch unit skips over the branch's CD region. Skipping past the CD regions in the A-stream is achieved by converting the branches to branch to its reconvergent-point instead of branching to its taken-target. This leaves the delinquent branch and its backward slice in the A-stream. The shorter A-stream runs ahead, pre-executes delinquent branches, and communicates the branch outcome to the R-stream.

For the astar ROI under consideration, the maparp branch b2 and the store s1 are control-dependent on the waymap branch b1. So when the A-stream attempts to pre-execute b1, it skips over the waymap branch's CD region thereby skipping past b2 and s1. Thus, SS2 is unable to pre-execute the maparp branch despite its delinquency. This highlights a key limitation of SS2 in its inability to pre-execute nested delinquent branches.

Furthermore, the write to the fillnum structure creates a loop-carried memory dependency between a store and the future instances of the eight different waymap branches. Since the store is in the CD region of the delinquent waymap branch, and the A-stream skips over this critical store, the pre-executed outcomes for the waymap branch could also be inaccurate and lead to mispredictions in the R-stream. Accounting for such stores is found to be helpful to achieve good pre-execution accuracy and performance speedups from our experiments.

Kumar et al. [KSC+21] propose a novel paradigm called Post-Fabrication Microar-

chitecture that tightly couples the superscalar core with a reconfigurable fabric. This way, custom pre-execution engines such as custom branch predictors and custom prefetchers can be manually customized to the application and synthesized on the reconfigurable fabric to address the superscalar core's microarchitectural inefficiencies.

In **Pruett's and Patt's Branch Runahead (BR) [PP21]**, lightweight dependency chains are extracted from the backward slices of delinquent branches. They use reconvergent point prediction to understand guard and affector relationships between branches. The dependency chain for the delinquent branch is terminated upon encountering its guard or affector branches. The constructed dependence chain is tagged with the guard or affector branch PC and its direction and new dependence chains are then constructed for the guard and affector branches. The tags are responsible for the linkage between dependence chains since, when a dependence chain completes execution, its PC and outcome are compared against all the dependence chains' tags and trigger one or more of them in the case of matches. This way, BR respects control-flow in the backward slices of delinquent branches, despite not including branches in the dependence chains.

Chapter 3

Branch Runahead

Branch runahead (BR) is the most recent work in the area of branch pre-execution at the time of writing this thesis. This chapter performs a deepdive into BR and explores the challenges faced in its implementation and applicability.

3.1 Overview

BR constructs lightweight dependency chains for each static delinquent branch. Each triggered instance of these dependency chains can pre-execute a single dynamic instance of its delinquent branch. Linkages between the dependency chains are established to keep the pre-execution running continuously.

BR improves upon prior pre-execution techniques through its ability to handle branches in the backward slice of a delinquent branch. This is done by identifying the control and data dependency relationships between branches and using that information to create the linkages between dependency chains. Identifying control dependency relationships among delinquent branches helps BR handle scenarios wherein delinquent branches may be nested within other delinquent branches, thereby solving the limitation of SS2. Likewise, considering CIDD relationships between branches enables BR to closely mimic the relevant data movement expected in the main thread.

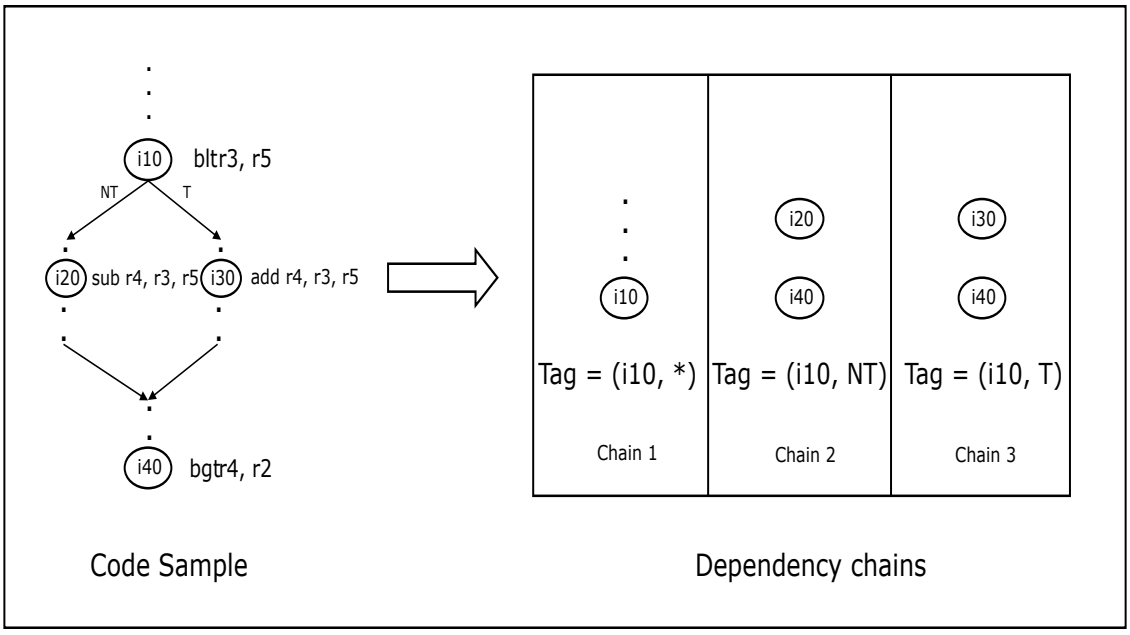


Figure 3.1: Sample BR dependency chains when delinquent branch i40 is CIDD on delinquent branch i10

The linkage is achieved by assigning tags to the dependency chains. BR uses these tags to dictate which dependency chain(s) are triggered next. By doing so, different dependency chains are linked serially to create an illusion of a pre-execution helper thread for those groups of chains. Different groups of chains may be formed if they are mutually control-independent and data-independent. Such mutually independent chains are unaffected by each other's progress thereby exhibiting a chain-level parallelism that enables BR to produce timely and accurate pre-execution.

Figures 3.1 and 3.2 show example scenarios wherein a delinquent branch is CIDD and CD on another delinquent branch respectively.

Looking at the resultant chains in Figure 3.1, BR constructs three different chains to handle this control flow scenario. Chain 1 results in the pre-execution of a dynamic instance of i10 and it is also tagged with itself. This indicates that the delinquent branch i10 is CI w.r.t other branches and thus may trigger future instances of itself. Chain 2 and Chain 3 each result in the pre-execution of i40 but are tagged with i10 and a specific direction. Such

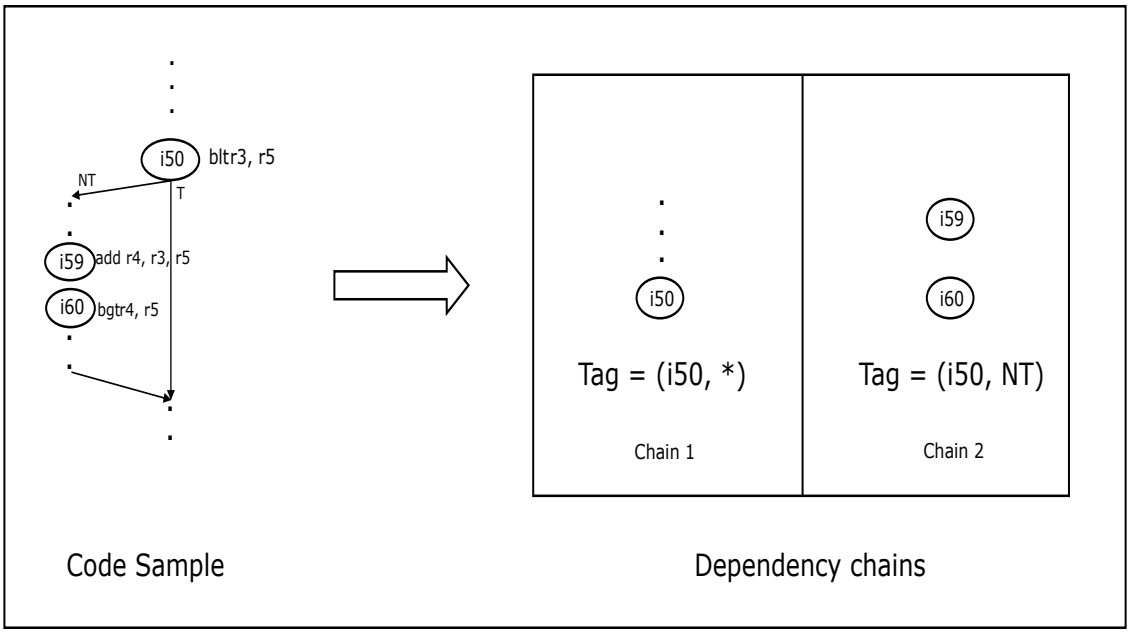


Figure 3.2: Sample BR dependency chains when delinquent branch i60 is CD on delinquent branch i50

linkages enable BR to correctly trigger the required dependency chain based on the observed outcome of i10. Likewise, in Figure 3.2 delinquent branch i60 is CD on the delinquent branch i50. The resultant Chain 2, is ensured to be triggered iff i50 resolves as NT. Thus, an instance of i60 is pre-executed only when its guarding branch i50 is observed to fall through. This way, BR manages to correctly capture the effects of a delinquent branch in the backward slice of another delinquent branch. While chaining helps BR to trigger the correct sequence of dependency chains, it could affect the timeliness of pre-execution of nested delinquent branches. This is because the nested delinquent branch can only be triggered when its guarding branch has resolved. We will explore this issue in a later discussion with an example.

The following subsections start out with detailing the dependency chain construction process and the notion of triggering the dependency chains. This is soon followed by our BR implementation methodology, a discussion on the performance of BR on the Makebound2() ROI, and finally highlighting the challenges BR faces.

3.2 Dependency Chain Construction

There are two key aspects involved in the dependency chain construction. One of it is the detection of control and data dependency relationships between branches.

BR, similar to SS2, uses merge (reconvergent) point prediction to identify the control-dependency and data-dependency relationships among branches. This is achieved by using a structure, that keeps track of instructions that are in the wrong path, called Wrong Path Buffer(WPB). When a misprediction is detected, the instructions that follow the misprediction branch are tracked in the WPB by performing a forward ROB walk from the mispredicted branch. As the processor recovers and instructions from the correct path retire, their PCs are checked for a hit in the WPB. A hit in the WPB indicates that the instruction was the first instruction to be observed in both the paths of the branch. This instruction is predicted as the merge point of the branch. By definition, any branch that appears between a branch and its merge point is said to be guarded by the merge-predicted branch. Similarly, any branch that sources data that originates from within the CD region (either of the taken/not-taken paths) of a merge-predicted branch is said to be affected by it. BR achieves this similar to SS2, by simply poisoning registers that are written in either of the paths. Additionally, to detect the flow of data through memory, the addresses written to by store operations on either path are buffered. Instructions beyond the merge point propagate the poison bits if they source a poisoned address, or register. During the propagation, any branch that sources a poisoned register is deemed as an affectee of the merge-predicted branch.

The second aspect of dependency chain construction is the process of extracting the backward slice of a delinquent branch. The dependence chain is initialized with a predicate computing instruction that evaluates the outcome of the targeted delinquent branch. Additional instructions are added to the dependence chain through a chain extraction process that involves a backward dataflow walk. To do this, BR buffers the last 512 retired instructions in a circular buffer called the Chain Extraction Buffer (CEB). Starting from an instance of the delinquent branch in the CEB, BR scans the CEB backward and adds instructions to

the chain if they are found to be a part of the computation that leads up to the delinquent branch.

To capture dataflow to the branches through store-load pairs, BR buffers the stores and their addresses as they are observed in the retirement stream. When a load instruction is added to the dependence chain, its address is checked for a match among the addresses buffered by stores, and any matching store instruction is also included in the dependence chain. Furthermore, the store-load pair is *assumed* to always conflict and is converted into a move operation. Promoting the store-load pairs into move ops ensures that the dependency chain does not contain any store operations. BR avoids control flow in the dependency chain by skipping past any branches that are not affectors or guards. When a guard, affector, or another instance of the targeted delinquent branch is encountered, the dependency chain extraction is terminated.

When the chain extraction process is terminated upon encountering an affector or a guard branch, the extracted dependency chain is tagged with the PC and direction of the guard or affector. BR then proceeds to extract the dependency chain for the guard or affector. When no guard or affector is detected, the chain extraction is terminated at the previous dynamic instance of the targeted delinquent branch, and in this case, the chain gets tagged with the targeted delinquent branch PC and a don't-care direction to indicate its control independence and data independence w.r.t. other branches.

3.3 Triggering

To kickstart the process of chaining, when the main thread mispredicts on a conditional branch, its PC and direction are checked with the existing dependency chains' tags. Any matching chain is triggered after its live-in values are synchronized with the main thread. To keep the chaining going, when a dependency chain results in a branch outcome being generated, its PC and direction are again checked against the existing chains' tags, and any

matching dependency chains are triggered. Such a chaining process enables BR to respect the CD and CIDD relationships learned so far.

BR makes use of multiple per-PC prediction queues to communicate the pre-execution outcomes from the dependency chains back to the main thread. At the time of triggering, each dependency chain reserves/pre-allocates an entry in its corresponding prediction queue. This enables the chains to execute and resolve out-of-order. Such a setup is required as the process of ordering dependency chains is not a trivial task as the chains are triggered similar to a data-flow fashion. The authors also propose multiple triggering policies for the dependency chains.

1. Chained initiation: As per this policy of triggering, the successor dependency chains are triggered only when the parent dependency chain has successfully executed and generated an outcome. This ensures that the chains are not speculatively triggered thereby simplifying the implementation aspect at the cost of delayed triggering of dependent chains.

2. Predictive initiation: In this case, as soon as a dependency chain is triggered, its outcome is immediately predicted and any chains that match the resultant tag (PC + outcome) are triggered speculatively. This policy is aimed at improving the timeliness of pre-execution as upon a mispredicted speculation, the correct chains would be triggered no later than they would have been in the chained initiation policy. However, this comes with the complexity of squashing the speculatively triggered dependency chains.

3.4 Methodology

We implemented the core-only version of Branch Runahead in our in-house, RISC-V, execution-driven, cycle-level, execute-at-execute simulator that models a superscalar core. Table 3.1 shows the superscalar core and memory hierarchy configuration used for running branch

Table 3.1: Superscalar core and memory hierarchy configuration. In case of partitioned resources, middle column indicates main thread's partition and right column indicates dependency chain's partition

branch predictor	64KB TAGE-SC-L	
pipeline depth	11 stages (fetch to retire)	
fetch/retire width	4 instr./cycle	4 instr./cycle
issue/execute width	8 instr./cycle	
execution lanes	4 simple ALU, 2 load/store, 2 FP/complex ALU	
ROB/PRF-FL/LQ/SQ	632/316/72/144	0/316/72/0
IQ	128	
L1I cache	32KB, 8-way	
L1D cache	48KB, 12-way, 3 cycles (1 agen, 2 hit)	
L1D prefetcher	next-N-line (N=2)	
L2 cache	1.25 MB, 20-way, 15 cycles	
L3 cache	3 MB, 12-way, 40 cycles	
L2/L3 prefetcher	VLDP (5.5 Kb)	
DRAM	100 cycles	

runahead.

The core's front-end stages, the PRF Freelist, and the LQ are partitioned into two halves. Half of the said resources are reserved for the main thread and the other half of the resources are reserved for the dependency chains. The IQ and execution lanes of the processor are not partitioned and freely shared by instructions belonging to both the main thread and the dependency chains. The SQ is wholly used by the main thread since the BR dependency chains do not include stores.

3.4.1 Astar case study

Before delving into the performance of BR in the Makebound2() ROI, let us begin with noting certain facets of the dependency chains that would be created by BR.

For simplicity, let us begin by temporarily assuming that no affector relationships are detected due to the loop-carried memory dependency at the construction of the dependence chain. This is a reasonable assumption to make because the store-load conflicts are highly dynamic anyway. The dependency chains for the waymap branches (b1, b3, ... b15) would

be tagged with their own PC indicating control independence w.r.t. other branches. This ensures that a waymap dependency chain continuously triggers the next dynamic instance of itself.

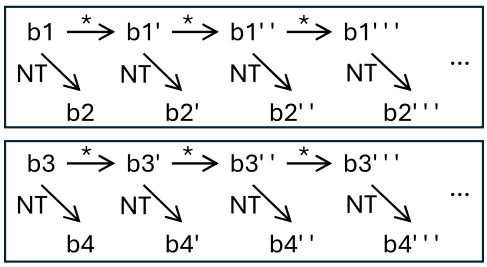


Figure 3.3: First two of eight chain groups in *astar*.

Next, the reconvergent point algorithm would capture the guarding relationship between any of the eight maparp branches and its most recent waymap branch. Thus, all maparp chains would be tagged with their corresponding waymap branch PC and a not-taken direction. Figure 3.3 serves as a visual guide for depicting the independence among two of the eight chain groups and the serialization within a chain group.

This dependency though correct, could affect the timeliness of pre-execution. For instance, when the chained initiation policy is used for triggering, the maparp's dependency chain can only be initiated when its corresponding waymap chain computes its predicate as not-taken. This delay in initiation could translate to lesser pre-execution accuracy as the main thread resorts to using the default branch predictor if there exists no pre-executed outcome at the time of fetch.

An alternative, the predictive initiation policy, is suggested by the authors of BR to circumvent this delay. The predictive initiation policy predicts a chain's predicate outcome at the time of its triggering. The prediction is made using a simple 3-bit saturating counter that reflects the bias of the chain. The predicted outcome is then used to speculatively trigger other dependent chains earlier than they would have been otherwise. For example,

in the case of Astar, predicting the outcome of the waymap's dependence chain at the time of its initiation could speculatively trigger the maparp's dependence chain. While it is true that this speculation comes at no extra cost other than squashing misspeculated dependency chains, highly unbiased branches such as the waymap branches are unbiased and very difficult to predict. Thus BR could benefit very little from such an optimization.

Furthermore, the loop-carried memory dependence explained earlier could pose additional challenges for BR. Because of this dependence, stores in an iteration of makebound2() can conflict with loads that feed future waymap branches in subsequent iterations. However, the highly dynamic nature of the worklists' contents implies that such conflicts are only occasional. However, the assumption made during dependence chain construction that store-load pairs always alias to the same address fails in this case. This could lead to reduced accuracy of pre-executed outcomes. Additionally, if we suspend the temporary assumption

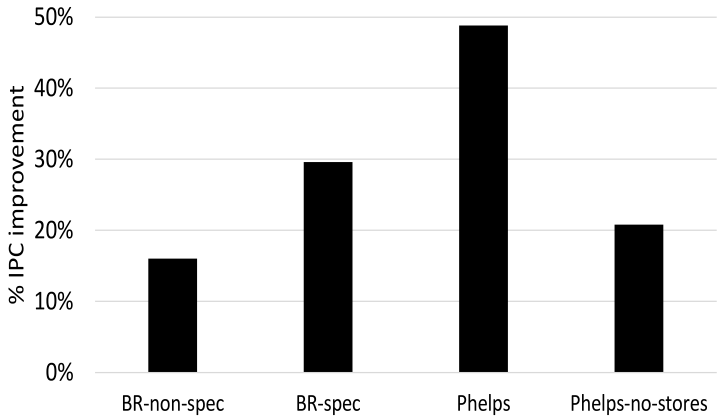


Figure 3.4: Phelps and Branch Runahead for *astar*'s top SimPoint.

made regarding affector relationships, a conflict between a store to the waymap structure and a load from the waymap structure could end up deeming a waymap branch to be affected by the maparp branch. This could cause the waymap branch's dependence chain to lose the control independence tag and now has to be triggered by the affector branches. This has the effect of serializing the dependence chains thereby losing any advantage gained through

control independence. Thus, while implementing our BR, we forcibly excluded stores from BR to help it.

Figure 3.4 compares the performance of BR with chained initiation (BR-non-spec), speculative/predictive (BR-spec) triggering, and Phelps on just the top-weighted *astar* SimPoint (in which makebound2() is prominent). BR-non-spec, due to the delayed triggering of maparp chains, achieves only a modest speedup over the superscalar baseline. This is improved upon by BR-spec by predicting the direction of the waymap chains at the time of triggering. BR-spec's inaccurate triggering of nested chains b2, b4, etc., and resulting rollbacks of their chain groups, is tolerated to some extent by having chain group level parallelism of 8, yielding a good speedup of 30%. Phelps as we will explore in Chapter 5, achieves a higher speedup (49%) with a simpler pre-execution paradigm: a single, self-contained, rollback-free (except for load violations) helper thread that is wholly free of branch prediction's limitations. The figure also features Phelps-no-stores where we forcibly exclude stores from the helper thread. Excluding the stores significantly impacts the performance of Phelps and this phenomenon could also play an important factor in BR not performing as well as Phelps for the makebound2 loop.

3.5 Additional challenges

The challenges faced by BR is not just limited to the inefficiency of speculatively triggering dependency chains and handling stores in the backwards slice of a delinquent branch.

- Independence among chain groups is exploited by selectively rolling back only the affected chain group, which is complex to implement in our experience. Selective rollbacks occur in two cases:
  - the main thread consumed an incorrect prediction from a chain. This scenario requires selective rolling back of all the chains belonging to that chain group.

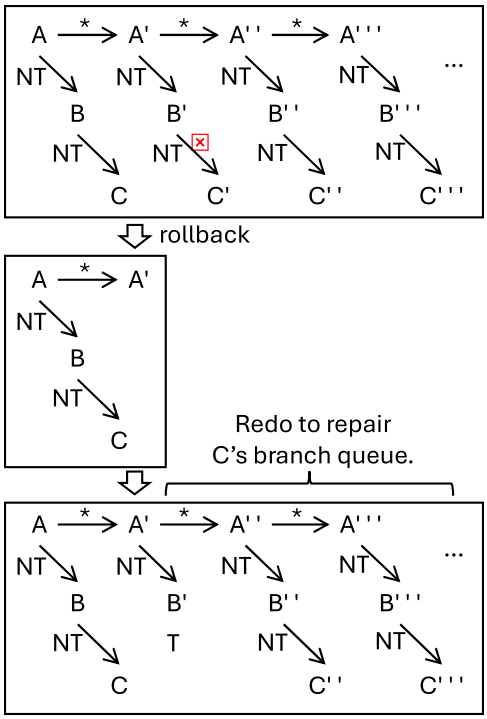


Figure 3.5: Branch Runahead: Rollback within an example chain group. B′ speculatively and incorrectly triggered C′. To repair Branch Runahead's branch queue of C, we must rollback to top-level branch A′, to retrigger A″, A‴, and their descendants.

– a parent chain speculatively triggered an incorrect child chain. this scenario requires rolling back to the top-level (self-dependent) chain in a chain group even if the flaw stemmed from a descendant of the top-level branch.

  Figure 3.5 depicts why this is the case. Under consideration is a chain group with three dependency chains that follow the triggering relationship A->B->C. Additionally, the top-level chain is self-dependent and will trigger another instance of itself. When chain B's outcome is predicted, and it speculatively triggers an instance of C, an entry is allocated in C's pred queue at its tail. If the prediction made was incorrect, simply squashing the speculative triggered instance of C does not suffice as the prediction queues are no longer in synch. To repair Branch Runahead's branch queue of C, we must rollback to top-level branch A′, to retrigger A″, A‴, and their descendants.

- BR is also challenged in the presence of nested loops. This is an artifact of a combination of the lack of program order among the dependency chains and the merge point detection algorithm.

  As introduced earlier, BR uses the WPB to buffer instruction PCs from the wrong path and looks for a match in the correct path instructions. This heuristic could yield multiple merge point predictions for the same branch. For instance, if we consider determining the merge point of the inner loop branch of nested branches, there could be two merge points predicted. First, the more obvious and instinctive merge point would be the instruction following the backward inner loop branch. In this case all the branches within the loop body will be deemed as being guarded by the loop branch. The header branch guards the loop branch and is itself deemed self-dependent. In this scenario, since the dependency chain for the header branch is tagged with a "*" direction, it keeps triggering instances of itself, thereby triggering multiple visits to the inner loop simultaneously. Processing the different visits simultaneously would require careful interleaving of inner loop branches belonging to different visits. However, there is no clear mechanism to interleave predictions from different visits.

  The second merge point candidate that we find being detected is rather interesting. The candidate happens to be the target of the backward loop branch. This happens in some dependency chain construction scenarios. For instance, when we over iterate on the inner loop branch, it is possible for the WPB to contain only instructions from the loop body if the inner loop branch was not predicted to fall through in the wrong path. In such a scenario, the correct path instructions would start from the instruction following the inner loop branch. Eventually, the correct path might reach the outer loop branch and start a new outer loop iteration. The first instruction that would hit in the WPB in this case would be the inner loop's branch target instruction from the next inner loop visit. A similar scenario can be constructed when the inner loop is under iterated. In these cases, the header branch of the inner loop is detected between

the inner loop branch and its merge point. BR would thus deem the header branch to be CD on the inner loop branch. This is partially true since the header bracnch is actually CD on the previous inner loop visit's loop branch falling through.

The header branch in this case is not deemed self-dependent but instead CD on the inner loop branch's NT direction or the header branch's T direction. The inner loop branch on the other hand is deemed to be CD on the header branch being NT or the inner loop branch being T. Such a chain construction serializes different visits to the inner loop thereby not requiring to interleave predictions. However, serializing the inner and outer loops misses the opportunity to decouple both the threads that Phelps exploits.

- As introduced earlier, only the dependency chains within a chain group adhere to program order by virtue of serialization between parent and child dependency chains. Chains across different chain groups are not in the program order and are thus triggered and reclaimed out-of-order. This necessitates that the physical registers belonging to the dependency chains are managed through reference counting [ARS03; MPV93]. So the physical register reclamation scheme uses a ROB-based freeing up of registers in the main thread's partition and a reference counting scheme for the registers in the BR's partition. This necessitates implementing both the register management scheme in the processor pipeline and reference counting is quite complex to implement by itself [Rot08].

- Another concern regarding BR is the lack of loop branches in the dependency chain. And since chains are synchronized only on mispredicts arising from the main thread, one such misprediction could trigger a chain with a small trip count. However, the lack of the loop branch in the dependency chain implies, BR keeps over-iterating and wasting resources down the wrong path until the next branch misprediction on the same branch.

Chapter 4

Custom Branch Predictors

4.1 Post-Fabrication Microarchitecture

In Post-Fabrication Microarchitecture (PFM) [KSC+21], Chanchal et al. make a case for custom microarchitectural components. Their study indicates that customizing microarchitectural components to individual workloads can yield large speedups. To make this customization possible and feasible, PFM proposes to couple a reconfigurable fabric with the superscalar core. This way, custom microarchitectural components such as custom branch predictors and custom data prefetchers, which can be broadly termed as custom pre-execution engines, can be synthesized on the reconfigurable fabric as needed.

Figure 4.1 depicts the substrate for PFM. To enable such a coupling between the superscalar core and the reconfigurable fabric, programmable interfaces called agents are used to observe or intervene at key stages of the core's pipeline through communication queues. The retire agent observes the core's retire stream for instructions of interest and communicates information about them back to the custom components. The fetch agent intervenes in the core's fetch stage to override the core's default branch predictions with predictions from the custom component. With such an organization, a custom pre-execution engine that closely mimics the behavior of the targeted delinquent branch can be synthesized on the recon-

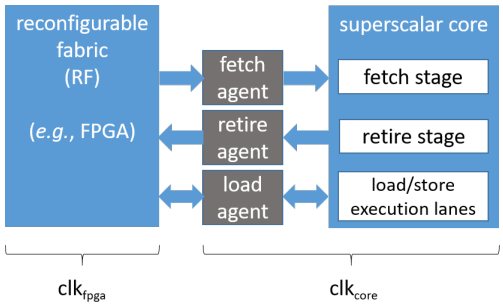


Figure 4.1: Substrate for Post-Fabrication Microarchitecture (PFM).

figurable fabric. Such pre-execution engines are shown to result in highly accurate branch outcomes being communicated to the superscalar core leading to large performance gains and energy reduction. The following subsections elaborate on two specific custom branch predictor designs of PFM.

4.2 Astar custom branch predictor

The Astar custom branch predictor is a customized datapath designed to pre-execute the delinquent branches in the ROI of the astar benchmark's top-weighted simpoint that was introduced earlier. It is designed to closely replicate the work achieved by the makebound2() and would be synthesized on the reconfigurable fabric of PFM to pre-execute the waymap and the maparp branches. Figure 4.2 shows the design for the astar custom branch predictor.

The figure shows the components of the design that would be synthesized on the reconfigurable fabric (shown as an FPGA) on the left. These include storage for the values that need to be snooped by the custom predictor like fillnum, yoffset, base addresses of the worklist, waymap, and maparp arrays. It also maintains three queues namely, the index queue, index1 queue, and the pred queue. Each entry of the index queue could contain a single index from the input worklist; hence, the size of the index queue dictates the number of loop iterations the design may run ahead. Each entry in the index queue would then generate 8 index1 with each requiring a maximum of two predictions. Thus, one entry in the index

queue allocates 8 entries in the index1 queue and 16 entries in the pred queue.

The components belonging to the agents are shown in the middle of the figure and the program's data structures, which the custom predictor loads data from, are shown on the right. Also, the figure is vertically demarcated into three regions each corresponding to "hardware" threads T0-T2. These are decoupled threads that make progress independent of the other threads and are only constrained by the different queues that they use to communicate with the other threads. Such a decoupled organization exposes the custom predictor to high memory-level parallelism as threads can go ahead and initiate cache misses for future work in its context without having to wait for the other threads to catch up. The ability to design a custom branch predictor with the insights about the ROI gives PFM the opportunity to extract such decoupling that was otherwise not exploited in the original program. The thread T0 is responsible for issuing loads to the program's worklist data structure through the load agent and populating the values into the index queue. The loaded values may arrive out of order and thus the load requests are tagged with pre-allocated ids (shown above the arrows) in their destination queues.

T1 is responsible for popping loaded values from the index queue and each of them generates the 8 index1 values across 4 cycles. Each of these values are then used to issue two loads, one to the waymap structure and one to the maparp structure, to eventually generate their waymap and maparp branch outcomes. Since these loads may arrive out-of-order they pre-allocate entries at the tail of the FPGA's pred queue. Even though the pre-allocation happens in the pred queue, the load IDs are obtained using the speculative head of the index queue due to a correlation between the entries in the index queue and the pred queue. Additionally, the index1 value is also stored in the index1 queue to indicate which index1 generated the waymap and maparp predicates for reasons explained later. When the load's values return, rather than storing the actual value in the pred queues, the waymap and maparp predicates are evaluated using the loaded values and stored in the entry pointed by the load ID.

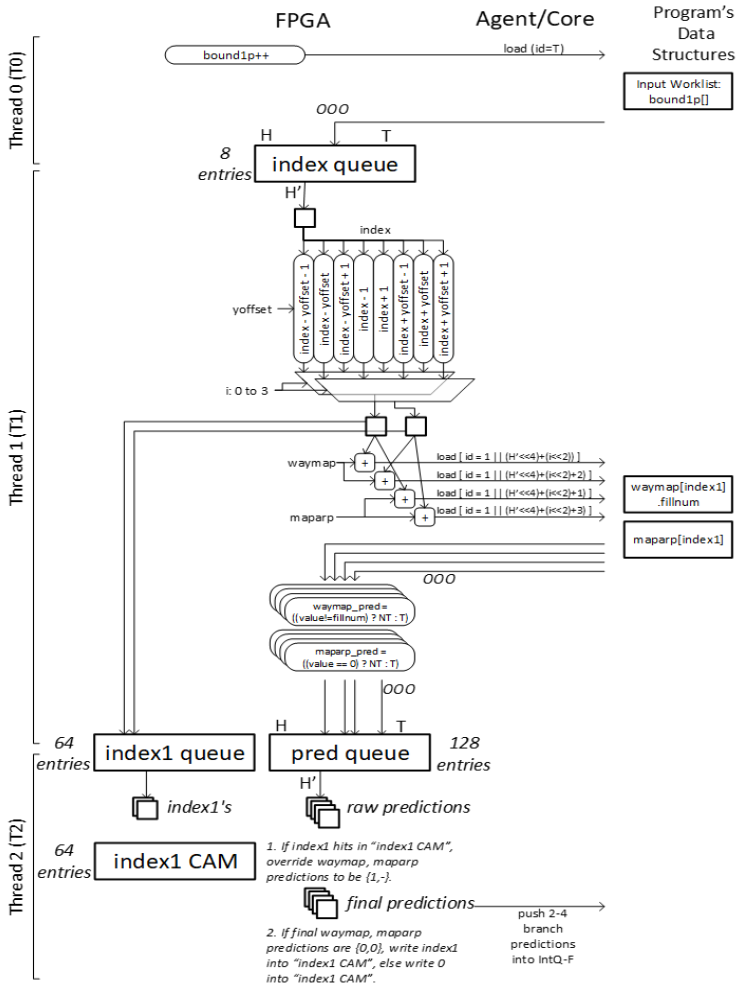


Figure 4.2: Astar custom branch predictor.

The thread T2 would consume raw predictions from the speculative head of the pred queue. These are termed raw predictions as they may have to be overridden due to the influence of the loop-carried memory dependency as explained later.

The custom design addresses the two challenges faced by the contemporary pre-execution paradigms. Firstly, it is capable of handling nested delinquent branches. This is achieved by virtue of applying predication-like principles in the custom design. This is because, regardless of the direction of the guarding waymap branch, the predicate outcome of the nested maparp branch is evaluated and populated into the pred queue. However, while pushing prediction outcomes into the fetch agent's intervention queue, the maparp outcome will be pushed if and only if its guarding waymap branch results in a not-taken (NT) outcome. In the case when the guarding waymap's outcome was taken (T) the maparp's predicate will simply be ignored at the pred queue's speculative head leading to a highly non-divergent pre-execution outcome stream. This was made possible through manually understanding the control dependency relationship between the maparp branch and its guarding waymap branch.

Secondly, the challenge of loop-carried memory dependence is also solved by inferring the existence of the store (line 28 in Figure 2.1). The custom predictor uses application-specific insight (that the presence of such a store is used to turn down subsequent visits to the same index1) to infer the store's existence and produce highly accurate branch pre-execution outcomes. This is implemented using a 64-entry CAM structure that keeps track of the instances of index1 that were visited in the scope of the custom branch predictor. Thus, when T2 processes raw predictions from the speculative head of the pred queue, it obtains the index1 that was used to generate the current raw predictions and searches for a hit in the index1 CAM. A hit in the index1 CAM implies that the index1 was recently visited in the speculative scope of the custom predictor and thus overrides the waymap prediction to taken (T) and pushes it into the intervention queue at the fetch agent and skips pushing the maparp outcome. When there is no hit in the index1CAM, the waymap outcomes are unconditionally pushed into the intervention queues, and maparp outcomes are only pushed

when their guarding waymap was deemed to be NT. If the final predictions were NT, NT for both the waymap and maparp branches, their corresponding index1 is pushed into the index1 CAM for making future inferences. As mentioned earlier, the index1 CAM only stores index1 values that are in the speculative scope of the custom predictor. Thus, when the retire agent observes the retirement of a waymap-maparp branch pair, the store is no longer speculative, and thus an entry is popped from the head of the index1 CAM.

While the branch pre-execution outcomes generated by the custom predictor design are perfect and do not diverge from the main thread's behavior, the main thread can roll back its fetch unit due to other sources of branch mispredictions, load violations, etc. To handle such rollbacks, the retire agent sends a "squash" packet to the custom predictor to notify it. The custom predictor then simply rolls back the work done by T2 to line up its predictions with the core's fetch unit. There is no need for squashing the work done by T0 and T1 as they are not speculative and thus unaffected by the squash. Because the index1 CAM now reflects inferred stores in the future, T2 cannot redo the work of converting raw predictions to final predictions as it initially had done. Therefore, final predictions are recorded in an additional queue and, after receiving a squash packet, those between the pred queue commit head (H) and speculative head (H') are replayed from this additional queue.

4.3 Bfs custom branch predictor

The PFM paper also shows a custom branch predictor that was designed for the region of interest in GAP benchmark suite's bfs benchmark. The bfs benchmark performs a breadth-first search on the nodes of a graph. It starts with a source node and starts examining its neighbors and "visits" neighbors that have not yet been visited and adds them to the frontier (worklist). The nodes added to the frontier in the current pass of bfs are processed similarly in the next pass. This process continues until there are no more nodes to be processed in the worklist.

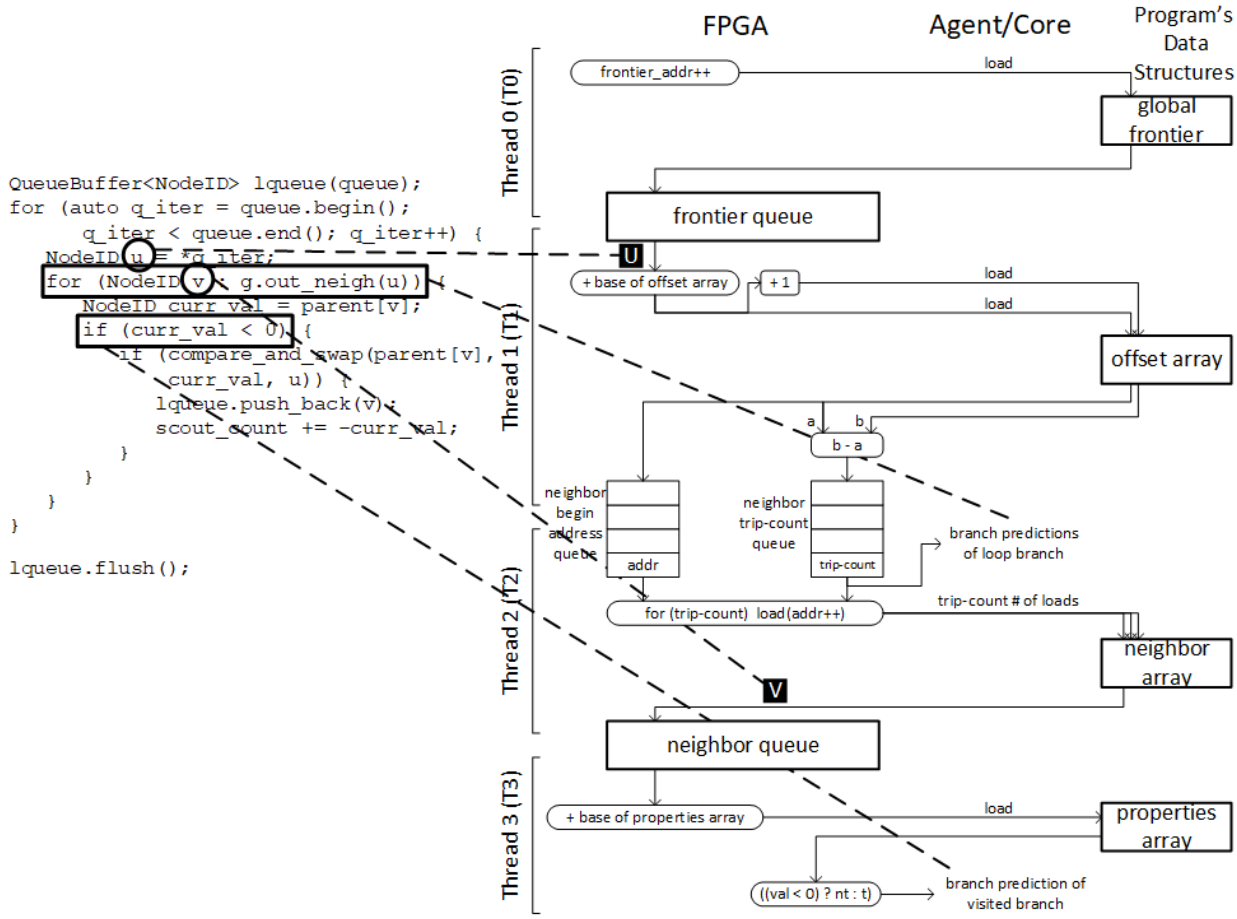


Figure 4.3: ROI and custom branch predictor for bfs.

The graphs in the GAP benchmark suite are expressed in a Compressed Sparse Row representation (CSR). In CSR representation, the graph is stored in multiple arrays namely the offset array, neighbors array, and the properties array. The neighbor array is a continuous list of neighbors of all the nodes in the graph. The offset array is indexed using the node id and it contains pointers into the neighbor array that corresponds to the first neighbor of the node. Thus, the list of neighbors of a particular node id can be found between the offsetarray[id] and offsetarray[id+1]. The properties array consists of the properties of the nodes in the graph such as weights, visitedness, parents, etc.

The left part of Figure 4.3 shows the code snippet from the TDStep() of the bfs benchmark. The code snippet corresponds to the actions performed in a single pass of bfs as

described above. Much of the code's actions are similar to the actions performed in the astar's ROI. The function consists of a doubly nested for loop. The outer for loop pops a node, u, from the worklist and evaluates the begin offset and the end offset of its subset of neighbors in the neighbor array. The inner for loop iterates from the begin offset to the end offset and loads the neighbor, v, contained in these pointers. Once the neighbor's index is obtained its property array is examined to check if the node has already been visited (by having a parent assigned). If the node has not yet been visited, the pass updates the visitedness of the node by writing into its parent field. This ensures that the same node is not visited again in a future iteration.

The inner for loop branch (marked by the first rectangle in the image) which iterates across the neighbors of the index u is a delinquent branch as the number of neighbors depends on the input graph connectivity and could be highly variable. The branch that tests the visitedness (second rectangle) of the neighbor index v is also delinquent because the worklist, and thereby the value of v are highly dynamic and may not be repetitive in nature. Also similar to the astar ROI, the visitedness branch is further affected by the loop carried memory dependence due to the update of visitedness described above. Thus, the challenges faced by the contemporary pre-execution paradigms are not just limited to the astar use-case and could be highly prevalent in graph workloads.

The custom branch predictor's design is shown on the right side of Figure 4.3 and uses a similar demarcation of components and their responsibilities as the astar custom branch predictor. The custom predictor snoops the base address of the frontier, and all the arrays involved to get started. It consists of four decoupled threads (T0-T3). The thread T0 issues loads into the frontier data structure of the program and fills up the frontier queue. The thread T1 pops an index u from the frontier queue and issues two loads into the offset array to obtain the begin offset and the end offset corresponding to u. Once these load values return, the begin address is pushed into the begin address queue for use by thread T2. Additionally, the difference between the end offset and the begin offset corresponds to the

trip count of the inner for loop. This value is pushed into the trip-count queue and is used to accurately predict the fall-through of the trip-count branch.

Thread T2 pops from the begin address queue and issues loads into the neighbor array to obtain the index of the neighbor (v). These values are pushed into the neighbor queue. T3 finally pops a node v from the speculative head of the neighbor queue and loads its visitedness by issuing a load to the property array. To handle the loop-carried memory dependence, we infer an outstanding store to a node's visitedness by examining prior neighbors in the neighbor queue. Any match with an index that lies between the commit head and the speculative head of the neighbor queue, causes T3 to not issue the load but directly push the outcome T for the visitedness branch. The predictions from the trip count queue and the visitedness predictions are interleaved to produce near-perfect branch predictions in the bfs ROI. Just as in the case of astar's custom branch predictor, the custom branch predictor for the bfs use-case also benefits from splitting the entire data path into decoupled hardware threads as the threads have the maximum degree of freedom for making forward progress and initiate cache misses early instead of waiting for the other threads to catch up.

4.4 Methodology

We use an in-house, RISC-V, execution-driven, cycle-level, execute-at-execute simulator to model the superscalar core, agents, and custom components. The superscalar core and memory hierarchy configuration is shown in Table 4.1. We use the top-weighted SimPoint [SPHC02] (100 million instructions) for each SPEC benchmark use-case. For the *bfs* use-case, from the GAP graph benchmark suite [BAP15], we skip the graph setup phase and run 100 million instructions of the breadth-first search (roadNet-CA graph [LK14]) or less if it completes in under 100 million instructions (com-Youtube graph [LK14]). In all results, the performance of the core with the custom component is normalized to the performance of just the core (which is at 0%).

Table 4.1: Superscalar core and memory hierarchy configuration.

branch predictor	64KB TAGE-SC-L [Sez16]
pipeline depth	10 stages (fetch to retire)
fetch/retire width	4 instr./cycle
issue/execute width	8 instr./cycle
execution lanes	4 simple ALU, 2 load/store, 2 FP/complex ALU
ROB/IQ/LDQ/STQ/PRF	224/100/72/72/288
L1I cache	32KB, 8-way
L1D cache	32KB, 8-way, 3 cycles (1 agen, 2 hit)
L1D prefetcher	next-N-line (N=2)
L2 cache	256KB, 8-way, 12 cycles
L3 cache	8 MB, 16-way, 42 cycles
L2/L3 prefetcher	VLDP (5.5 Kb) [SKB+15]
DRAM	250 cycles

In the experiments, various parameters of the custom component (C, W, and D, below) and Agents (Q and P, below) are varied to explore their performance impact. These parameters and their notation are described below.

- clk$C$_wW: C is the factor by which the RF-synthesized custom component's clock frequency is slower than that of the core (CLK_{CORE} / CLK_{RF}). W is the custom component's superscalar width. The custom component can generate up to W predictions and push/pop up to W packets into/from the communication queues, in a given CLK_{RF} cycle.

- delay$D$: D is the pipelined execution latency of the RF-synthesized custom component, in CLK_{RF} cycles (*e.g.*, for configuration clk4_w4, delay8 would mean 8 CLK_{RF} cycles or 32 CLK_{CORE} cycles).

- queue$Q$: Q is the size of the Observation and Intervention queues in the Agents.

- port$P$: P describes which Physical Register File (PRF) ports the Retire Agent can contend on (*e.g.*, portLS means the Retire Agent can opportunistically use the PRF ports owned by the two load/store execution lanes, while portLS1 limits sharing to only one load/store execution lane's PRF ports).

4.5 Results

4.5.1 Astar

Figure 4.4 shows the sensitivity of the performance of the custom predictor to a combination of its frequency differential with the superscalar core and its width (C and W parameters in clk$C$_wW) which combine to determine the custom predictor's bandwidth.. The first bar shows the speedup potential of perfect branch prediction being 162% over the baseline. With a custom predictor as fast as the core, a predictor width of 1 is sufficient to attain such speedups. This is because we eliminate most of the mispredictions as evident in the reduction of MPKI from 31.9 to a mere 1.04. In fact, the clk1_w1 configuration performs slightly better than perfect branch prediction which can be attributed to a prefetching effect for the core's redundant loads.

When the core is twice as fast as the custom predictor as in the clk2_w1, we observe a reduction in the speedup compared to earlier but the speedup over the base line is still quite significant. However, the lost speedup can be regained by increasing the superscalar width of the predictor to 2.

As the frequency differential gets even larger, with lower predictor width, we observe further reductions in speedup and even a slow down in one case. This is because at low bandwidth, the fetch unit stalls waiting for predictions from the custom predictor. However, we observe that the low frequency of the custom predictor can be sufficiently compensated by making the custom predictor wider. For example, the clk4_w4 configuration performs as well as the clk1_w1 configuration.

4.5.2 Bfs

Experiments were performed with two input graphs obtained from [LK14], namely Roads (roadNet-CA) and Youtube (com-Youtube). The analysis that follows refers to only the Roads input but similar trends can be observed in the Youtube counterpart. The first three

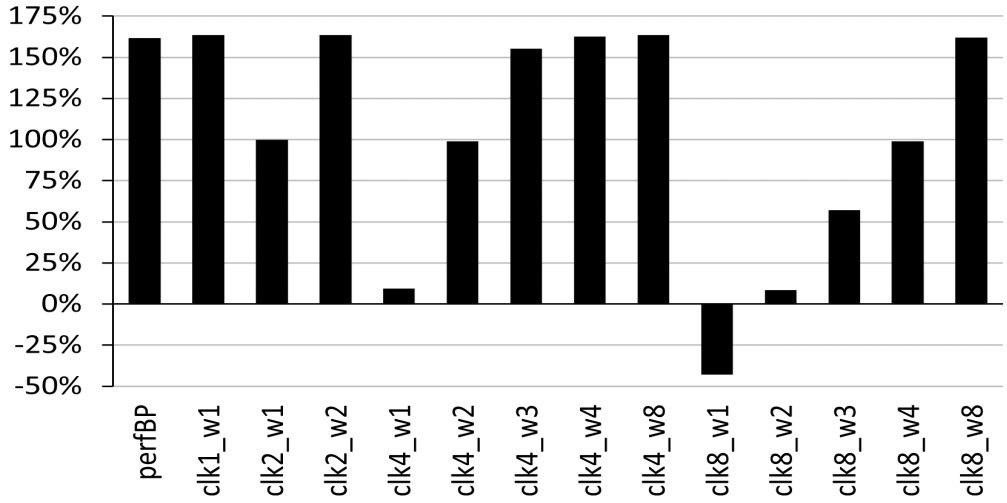


Figure 4.4: Speedup of the custom *astar* branch predictor for different C and W parameters. (All configurations: delay0, queue32, portALL; 8-entry index_queue.) *perfBP* is perfect branch prediction.

bars of Figure 4.5 show that both cache misses and branch mispredictions are debilitating. Importantly, both need to be attacked simultaneously. Perfect branch prediction alone yields only 11% speedup. Perfect data cache alone yields a significant 152% speedup, yet this is only a fraction of what the two together achieve: 426%. The trends for the sensitivity study of Bfs' custom predictor are quite similar to that of Astar's custom predictor. *Bfs*'s custom component achieves up to 125% speedup, due to (1) exploiting high MLP both within and among decoupled "threads" T0–T3 and (2) reducing branch MPKI from 19.1 to 0.5. Additional sensitivity studies concerning the intervention queue sizes, delay of the custom predictor component, and scope of the predictor were performed for both the custom pre-execution engines and listed in [KSC+21].

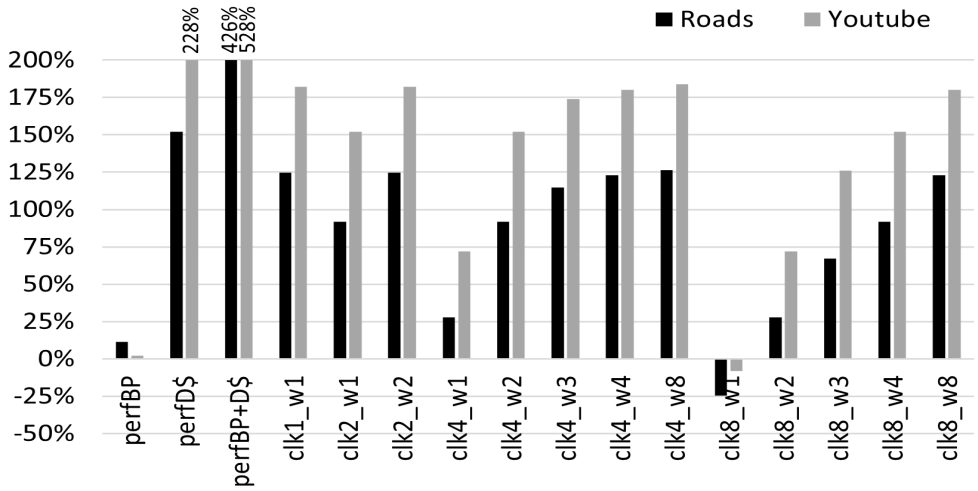


Figure 4.5: Speedup of *bfs*'s custom component, for different C and W parameters. (All configurations: delay0, queue32, portALL; 64-entry frontier, begin-address, trip-count, and neighbor queues.) *perfBP*, *perfD$*, and *perfBP+D$* are perfect branch prediction, perfect data cache, and both.

Chapter 5

Predicated Helper Threads

5.1 Overview

Phelps is a fully automated hardware-only technique that leverages branch predication within the helper threads to achieve timely and highly accurate branch pre-execution.

It starts out with identifying loop(s) containing delinquent branches and constructs helper thread(s) for the loops. The helper thread starts with just the delinquent branches, loop branches, and a header branch (if nested loops). Later, across several iterations of the loop, Phelps adds any other instructions needed to accurately pre-execute the delinquent branches. These are instructions that directly or transitively feed data to the delinquent branches.

Growing the backward slice of the delinquent branch is made possible by simply adding the direct producers of instructions already present in the helper thread. To capture data dependencies that persist through memory, a queue of the most recent addresses written to by store instructions and their PC is maintained. When a load instruction that is already included in the helper thread retires, it looks for a match on its address in the store address queue. Upon a match, the candidate store is included in the helper thread, and eventually its backward slice too. The backward slice of the instructions in the helper thread is grown across several iterations as done in [CHA+15]. This helps in avoid needing large structures

that can buffer instructions to perform dependency analysis.

Through backward slicing the helper thread consists only of instructions that feed data to the delinquent branches, the delinquent branches themselves, and the loop branches. Phelps converts all the delinquent branches into predicate producers within the helper thread. Thus they are no longer treated as control-flow within the scope of helper threads. The loop branch is the only control-flow branch included in the helper thread to determine when to terminate the pre-execution.

Phelps makes use of per-PC prediction queues to communicate the pre-execution outcomes from the helper thread to the main thread. The predicate-producer instructions all unconditionally push their outcome into their corresponding prediction queues as they retire. This is enabled by operating the pointers of the prediction queues in a lockstep manner. In other words, the head, the tail, and the spec head pointers are shared by all the prediction queues that belong to the helper thread.

The head pointer is incremented only when an instance of the loop branch is observed in the main thread's retire stream. The tail pointer is incremented whenever an instance of the loop branch is retired from the helper thread. The spec head pointer is kept in sync with the main thread's fetch unit and is incremented whenever the loop branch is fetched by the main thread. These pointers are essentially pointing to a specific iteration of the loop in the prediction queue. Such a prediction queue reserves an entry for every delinquent branch in every iteration of the loop. It is the main thread's fetch unit that consumes only the required predictions from the queues by using the fetch PC. Such an arrangement makes it possible for the helper threads to eagerly pre-execute nested delinquent branches and push their outcomes unconditionally. Wrongly pushing an outcome into the prediction queue does not leave the prediction queues in an incorrect state, unlike BR's prediction queues. The main thread's fetch unit takes care of the necessary corrective action in this case by simply not consuming the stale prediction from this iteration.

Figure 5.1 shows an example of this operation for the astar example we introduced in

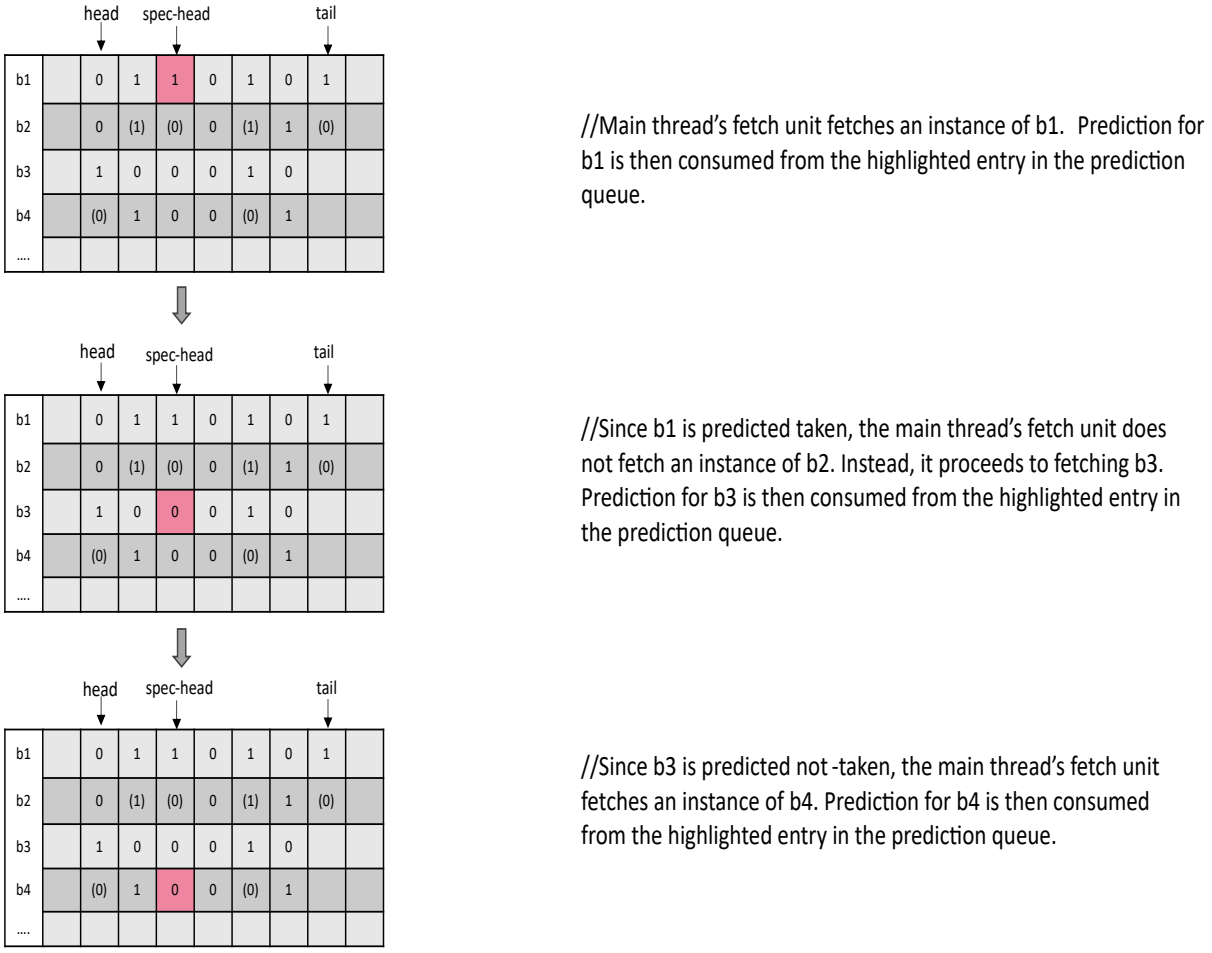


Figure 5.1: Example showcasing the interaction between the main thread's fetch unit and the prediction queues

Chapter 2. Only the first four of astar's sixteen delinquent branches are shown for brevity. Each column in the prediction queue corresponds to a loop iteration and each row is reserved for a delinquent branch. The helper thread deposits outcomes (0: not-taken, 1: taken). Notice that some predictions for b2 and b4 are shown in parentheses. This is just to highlight the fact that these predictions are not consumed by the main thread because the predictions consumed for their guarding branches b1 and b3 are 1 (taken) causing the main thread to not even fetch the corresponding instances of b2 and b4. This is visualized in the snapshots of the prediction queues in Figure 5.1. The first snapshot shows that the main thread's fetch unit is ready to process a new iteration of the makebound2() loop. Since it encounters the first

waymap branch b1, it finds a prediction queue allocated and thus consumes the prediction from its prediction queue at the spec-head pointer. Since the consumed prediction was taken, the fetch unit is redirected to b1's target skipping over b2's instance. Once b3 is fetched by the main thread, it is found to have an allocated prediction queue and thus overrides the default branch prediction with the prediction obtained from the prediction queue. Since the resultant prediction was not-taken, the main thread's fetch unit resumes fetching from the next instruction. This way, the main thread's fetch unit skips over pre-execution outcomes that ought to have been nullified, thereby implicitly predicating it.

In the event of a recovery initiated in the main thread, it is important to synchronize the pointers of the pred queue to ensure that the predictions produced by the helper thread line up with the main thread's fetch stream. For this reason, the spec head pointer of the pred queue is checkpointed at the fetch of branches in the main thread. Simply setting the spec_head pointer to the checkpointed spec head pointer (in the case of accelerated recovery using branch checkpoints) or head pointer (in the case of recovery from the head of the AL) enables us to reuse the already pre-executed outcomes instead of generating them all over again. This is even true for a branch misprediction in the main thread that stemmed from a helper thread outcome.

In fact, this underscores another subtle but important benefit. Though rare, the helper thread can generate an incorrect outcome for an instance of b1 in the astar example (e.g., due to losing a store's update in the helper thread's tiny data cache before the main thread retires its corresponding store). If that instance's incorrect outcome was "taken", the main thread initially didn't consume the corresponding b2's prediction. But because all predicate producers are eagerly executed, fortunately, the prediction for b2 exists nonetheless and can be revisited by the main thread. Thus, the helper threads are only affected by recoveries in their own thread and function independently of the other threads. To support pre-executing delinquent branches from both the helper threads, each helper thread has its own set of pred queue pointers.

Our approach is termed delinquent loop pre-execution with predicated helper threads because of using predication principles within the helper threads to pre-execute loops. Handling the challenge of nested delinquent branches was made possible through the lockstep-operated prediction queues. This is a form of implicitly predicating the nested delinquent branches as the main thread skips consuming the nullified ones. Explicit predication support is needed within the helper thread to handle only stores in the helper thread. Stores that are included in the helper thread are predicated on their most immediate guarding producer. Doing so enables us to nullify stores that are not in the correct control flow path. To achieve this, delinquent branches in the helper thread are converted to predicate producers. More on this in the next paragraph.

It is important to note that these stores are redundant future/pre-executed versions of stores that will be performed by the main thread. They are purely local to the helper thread's private mini cache and don't write into the data caches or memory upon committing. They only serve to help forward values to future loads within the helper thread[PSR02]. This enables the helper threads to run ahead of the main thread without waiting for the main thread to catch up and, simultaneously not clobbering memory. The helper thread stores have to be predicated if they are found to be CD on a delinquent branch. We achieve this by adding an additional source operand for the store instruction. This operand corresponds to the predicate destination register of its immediate guarding delinquent branch. If the store is nested inside more than one delinquent branch, however, one predicate is insufficient. To compensate, each predicate producer also has one predicate source operand, which gets linked to its immediate guarding predicate producer. Thus, if a store is nested inside multiple delinquent branches, it becomes transitively dependent on all of their predicates. Additionally, the store queue entry is augmented with an additional field for its computed predicate. When load instructions are issued from the helper threads and look up the store queue, in addition to matching on addresses, the store's predicate should have evaluated true to be a candidate for forwarding the value.

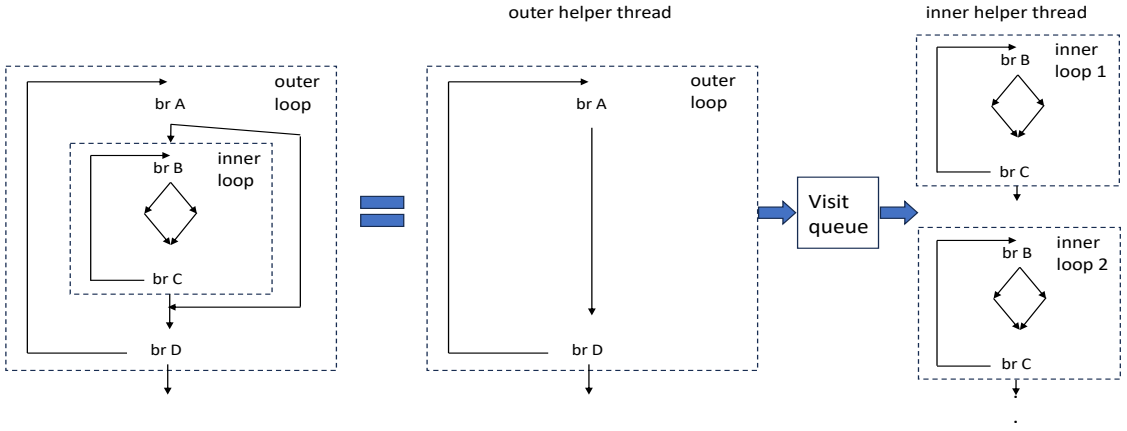


Figure 5.2: Decoupling of nested loops into outer helper thread and inner helper thread

In the case of delinquent branches being nested within deeper loops, Phelps automatically decouples the deepest two levels of loops thereby creating two helper threads referred to as the outer thread and the inner thread. The outer thread is responsible for forking multiple visits to the inner thread and this decoupling, as explored in the work of PFM, enables the outer thread to run far ahead into the future instead of waiting for the inner thread to catch up. Figure 5.2 shows how this decoupling is achieved.

This decoupling exposes much more memory-level parallelism (MLP) in the helper threads as cache misses from a future iteration of the outer loop can be initiated all while the inner loop is working its way through the current iterations. As the outer thread is responsible for producing the live-in information required for triggering a visit of the inner thread, such decoupling requires the live-in values to be buffered in a structure we call Visit queues. Further details about the visit queues and the implementation details for Phelps are given in the following sections.

5.2 Implementation

This section details the implementation aspects of Phelps and here is an outline of its contents.

Subsection 5.2.1 defines the notion of an epoch in the context of Phelps. Subsection 5.2.2 describes how Phelps identifies delinquent branches and the loops that contain them. This is followed by a detailed description of the process of helper thread construction in Subsection 5.2.3. Next we learn about how Phelps uses Control Dependency FSMs (CDFSMs) to identify the immediate guarding branch in Subsection 5.2.4. The following Subsection 5.2.5 provides details of the Helper Thread Cache (HTC) that stores the helper thread instructions. Once the helper thread instructions are loaded into the HTC, the helper thread is ready to be triggered and this process is explained in Subsection 5.2.6. The following Subsection 5.2.7 briefly talks about the terminating conditions of helper threads. Subsection 5.2.8 follows up with explaining the modifications needed in the superscalar core to implement Phelps. Subsection 5.2.9 talks about the different complex scenarios that were omitted from Phelps but provides a brief solution for each of these. Subsection 5.2.10 lists the conditions that may render a helper thread ineligible for pre-execution. Subsection 5.2.11 finally talks about the inability of Phelps in handling delinquent branches that are nested within a function call.

5.2.1 Epochs

Phelps continuously looks for opportunities to pre-execute hard-to-predict branches. It constructs and deploys predicated helper threads entirely in hardware (no compiler support nor instruction set architecture changes).

The entire process is pipelined into three stages, wherein each stage runs for an epoch of 4 million retired instructions of the main thread. Throughout this continuous process, the information about delinquent branches and loops gathered in epoch N is used to construct new helper threads, or revise existing ones, in epoch $N+1$, which are then available for

PC	misp	inner loop valid	inner loop branch	inner loop target	outer loop valid	outer loop branch	outer loop target
0x11b98	5760	1	0x11bfc	0x11b80	1	0x11c0c	0x11b60
0x11be0	7796	1	0x11bfc	0x11b80	1	0x11c0c	0x11b60

DBT index	Misp
1	7796
0	5760

loop branch	loop target	is nested loop	inner loop branch	inner loop target	delinquent branch list	misp
0x11c0c	0x11b60	1	0x11bfc	0x11b80	0x11b98, 0x11be0	13556

Figure 5.3: From top to bottom: DBT, DBT-Max, and BBT.

deployment in subsequent epochs *N+2* and greater. In a given epoch, all three stages are simultaneously active, with the second and third stages working with outputs from their feeder stages.

5.2.2 Identifying loops containing delinquent branches

To identify the most delinquent branches in the current epoch, we use the Delinquent Branch Table (DBT) and Delinquent Branch Table - Max (DBT-Max), similar to the tables used in Slipstream 2.0 [SCR20], albeit tracking only branches.

The contents of these tables are shown in Figure 5.3. The DBT maintains information about every conditional branch that was mispredicted during the epoch. The DBT is searched the PC of the branch instruction and each entry consists of a misprediction counter, and PC bounds of the inner loop and outer loop that enclose the branch, if applicable. When a conditional branch retires, if it was mispredicted by the core's branch predictor, it increments its misprediction counters accordingly. To train the inner and outer loop PC bounds, the

retirement unit keeps track of the PC and target PC of the most recently retired backward branch. When a conditional branch is retired, if its PC is between the backward branch's PC and target PC, the backward branch may replace the inner or outer loop fields of the branch's entry; which is replaced, if any, depends on comparisons among loops' PC bounds; in other words, the two loops whose bounds are closest to the branch are kept, and sorted as inner (tightest loop bounds) and outer (next tightest loop bounds).

The DBT-Max ranks the most delinquent branches observed so far. Each entry contains the DBT index of the corresponding delinquent branch, along with its misprediction count for maintaining delinquency ranking. Incrementally updating rankings in the DBT-Max, as mispredicted conditional branches retire and update their counts in the DBT, avoids having to scan the DBT at the end of the epoch to determine the top delinquent branches.

The Loop Table (LT), also shown in Figure 5.3, consolidates information about *outermost* loops and the delinquent branches contained within them. It is populated at the end of the epoch, as follows. A pass is made through DBT-Max. During the pass, each delinquent branch in DBT-Max that clears a threshold of 0.5 MPKI (i.e., 2000 mispredctions for the 4 million instruction epoch) creates or updates a LT entry for its *outermost* loop branch (gotten from the DBT). The delinquent branch augments the loop's aggregate misprediction count with its misprediction count and adds itself to the loop's delinquent branches list (a bit vector representing DBT-Max entries); if the delinquent branch has both an outer and inner loop in its DBT entry, then it adds the nested inner loop information to its outermost loop's LT entry. At the end of this pass, the LT contains information about the outermost loops (either *outer+inner* for nested loops or *inner* only) that contain the most delinquent branches. The DBT and DBT-Max counters are then reset in preparation for the next epoch.

5.2.3 Helper thread construction

If Stage 1 identified multiple delinquent loops in the previous epoch, we pick the most delinquent loop among them that doesn't have a helper thread in the Helper Thread Cache

(HTC). Stage 2 does helper thread construction for this loop in the current epoch. If the chosen loop is a nested loop, two helper threads are constructed at the same time: an outer thread and an inner thread. If the chosen loop is not a nested loop, just an inner thread is constructed. Hereafter, we refer to three helper thread types: outer-thread, inner-thread, and inner-thread-only.

Phelps uses backward slicing to construct the helper thread slice. Extracting backward slices requires having both source and destination register specifiers of instructions post-retirement. The ROB has destinations but not sources. Although it is possible to augment the ROB [HP15], we opted for a different approach. In any case, the full instructions will eventually be needed, to write the helper threads into the HTC. Therefore, as a preliminary step, all instructions in the loop are collected in the Helper Thread Construction Buffer (HTCB) as they are fetched and decoded by the main thread. Only a subset of them will be selected for inclusion in the helper thread as explained next.

To kick-start this construction process, the helper threads are initialized with the "seed" instructions. The seed branches are classified into three types as explained below. The following classification justifies the inclusion of such branches in the helper thread and also dictates the actions to be taken while executing these branches.

- Delinquent branches: These are the hard-to-predict branches that are identified using the DBT and DBT-Max tables. These branches are required as seeds in the helper threads since they are targeted for pre-execution. The delinquent branches are converted into predicated producers and may predicate other instructions in the helper thread. Their outcomes are also unconditionally pushed into the pred queue for later consumption by the main thread.

- Loop branches: These are the backward branch(es) that are identified in the delinquent branch and loop detection phase. The loop branches may themselves be delinquent but are nevertheless required in the helper thread to keep them running continuously and to know when to terminate the loop. These branches do not predicate any instruction

in the helper thread since they are backward branches. Mispredicting these branches would only result in a local squash within the helper thread. Only delinquent loop branches may push into the pred queue.

- Inner loop's header branch: The header branch for the nested inner loop branch is identified as the first branch within the outer loop that precedes the inner loop's branch target and branches beyond the inner loop. This header branch may be identified by looking for the above criteria while walking backwards from the inner loop's target pc in the main thread's retirement stream. These branches are required to be included as a seed branch in order to skip the triggering/forking of the inner helper thread when the header branch resolves as taken. These branches always push their predicate into the visit queue.

For inner-thread-only and inner-thread, the seeds are the inner loop's delinquent branches and backward branch. For outerthread, the seeds are the outer loop's delinquent branches, its backward branch, and the inner loop's header branch that guards the visit to the inner loop. The above branches are seeded into their respective level of the loop. This is done by comparing the branch PC against the backward loop branches and their target information obtained from the previous phase. Once the seeds are planted in the appropriate helper thread, Phelps starts growing the helper thread by adding the backward slice of the instructions already included. The backward slice of an instruction refers to the set of instructions that influence the flow of data to the instruction. To assist in this process, a last-producer table(LPT), shown in Figure 5.4, is maintained. The LPT has as many entries as there are logical integer registers in the ISA. As instructions retire in the main thread, they index the table using their logical destination specifier and deposit their PC into it. The table thus keeps track of the most recent instruction to write the register.

When an instruction that is already in the helper thread is observed in the retire stream, its producer instructions' PCs is determined by accessing the LPT using the logical source operands. Any producer instruction that is not already in the helper thread will be added

if they are within the bounds of the loop. This way the height of the backward slice(s) in the helper threads grows by one every iteration , eventually reaching a point where the producers lie outside the loop. By using such an incremental approach and training across several iterations, we avoid the need to buffer the retired instructions to extract the backward slices [CHA+15]. [KR04] uses a similar algorithm based on implicit back propagation to prune ineffectual instructions from the A-stream of Slipstream[SPR00].

The producer instructions observed in the backward slicing process which happen to fall outside the loop boundaries are deemed as live-ins and thus their logical destination specifier is added to the live-in register set. Phelps also has the ability to train and trigger helper threads from the middle of a loop rather than waiting for an entry into the loop. This implies that live-ins that are clobbered within the loop need to be identified and captured (since we might not train across different visitations to the loop). We make an observation that producer instructions which are logically after a consumer instruction, clobber a live-in to the loop. Thus, while looking up the last producer table, if the producer is logically after the consumer it is also considered as a live-in register and added to the live-in register set. A helper thread may have one or two live-in register sets, depending on the helper thread type:

- Inner-thread-only or outer-thread: One live-in register set, for live-ins coming from the main thread.
- Inner-thread: Two live-in register sets, for live-ins coming from the main thread and the outer-thread.

Similarly, to capture data dependencies that persist through memory, a 16-entry queue of the most recent addresses written to by store instructions and their PC is maintained (also shown in Figure 5.4. Only stores whose PCs are within the loop's PC bounds are captured in this store queue. When a load instruction included in the helper thread retires, it looks for a match on its address in the store address queue. Upon a match, the candidate store is

	PC
0	0x17588
1	0x17590
2	0x175b4
...	...
...	...
30	0x1760c
31	0x17594

	Address	PC
0	addrA	0x175ec
1	addrB	0x175f8
2	addrC	0x175fc
...	...	...
...	...	...
14	addrP	0x17694
15	addrQ	0x17698

Figure 5.4: Structures used for backward slicing; left: Last Producer Table (LPT); right: Store Address&PC Queue

included in the helper thread, and eventually its backward slice too. At the end of the epoch, the helper thread is considered to be sufficiently grown and passed onto the next stage.

5.2.4 Learning immediate guarding predicate producers

Before diving into the next stage of forming predicated helper threads and learning about the helper thread cache, it would be useful to learn how Phelps detects CD relationships between the predicate consumers (delinquent branches and stores) and predicate producers (delinquent branches). Phelps learns of these control dependencies through a novel iterative approach involving an FSM matrix shown in Figure 5.5 and uses them to generate predicated helper threads.

The Control Dependency FSM (CDFSM) matrix, shown in Figure 5.5, is a matrix of simple homogenous FSMs. Each FSM in the matrix can assume any of the four possible states shown. The states each correspond to the different control dependence relationships inferred until this point. A matrix of CDFSMs is required to understand the CD relationship between the predicate producers and the predicate consumers in the loop. Each row in the

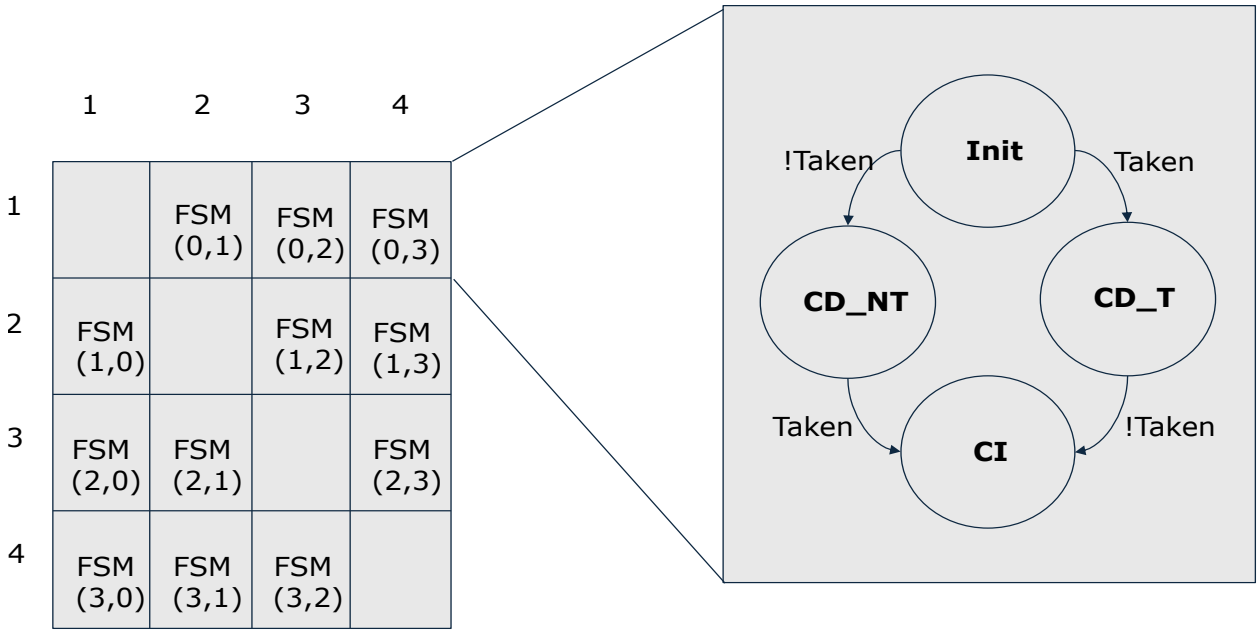


Figure 5.5: Control Dependency Detector State Machine Matrix (left), Control Dependency State Machine (right).

FSM matrix is reserved for a predicate consumer within the loop and each column in the matrix is reserved for a predicate producer in the loop. Thus, the FSM matrix is indexed along the columns by the unique ID allocated to the predicate producer branch that is observed in the loop, while the rows of the FSM matrix are indexed by the ID allocated to the predicate producer.

All FSMs in the matrix start in the init state and the state transitions are based on the simple heuristic that, an instruction is deemed control-dependent on a branch if and only if it is observed in only one of the control-flow paths of the branch. In other words, an instruction is control independent w.r.t. a branch if it is observed in both the paths of the branch.

To identify immediate guarding branches a branch list that contains all the branches observed in the current iteration and their direction is maintained. Every time an instruction in the loop retires, the FSM corresponding to the instruction and the most recent branch in the branch list is trained. If the instruction is already deemed control independent of this branch, the next recent branch in the list is selected for training. This way it is eventually

possible to find any instruction's immediate guarding branch.

To ensure that instructions only establish valid dependencies with branches that are statically before them and from within the same iteration of the loop, the branch lists pops entries that lie between the backward loop branch and its target every time the loop branch is retired. After training across several control flow scenarios, the FSMs in the matrix are indicative of the control dependency relationships that exist between the predicate consumers and their immediate guarding predicate producer.

5.2.5 Helper thread cache

After constructing a helper thread and learning immediate predicate producers

1. The instructions selected for inclusion in the helper thread are retrieved from the Helper Thread Construction Buffer(HTCB). The HTCB is needed for two reasons:

    - The process of backward slicing during helper thread construction requires the instruction's logical source operands which are not available in the ROB at retirement. So instructions that belong to the loop are fetched into the HTCB after the decode stage.

    - Since the helper thread construction process only keeps track of the PCs of instructions that are to be included in the helper thread, the actual instructions have to be somehow fetched into the HTC. While these can be opportunistically fetched whenever the main thread is not utilizing the instruction cache, the HTCB already buffers instructions for the backward slicing process and provides a natural way to access these instructions.

2. The Delinquent branches in the helper thread are converted to predicate producers. At the beginning of each epoch, each predicate producer is assigned a unique logical destination predicate register. This is same as the id used to index into the CDFSM

matrix. These ids start from 1 and is incrementally assigned to every predicate producer. The predicate producer's predicate destination operand can be encoded in bits 11:7 of the instruction, where rd resides for other instruction formats, because the branch's immediate is not needed.

3. Each store and predicate producer is given an extra source operand, for its logical source predicate register. One additional bit encodes the direction of the control dependence (taken/not-taken). The CDFSM matrix indicates which immediate predicate producer the store or the predicate producer depends on, and the direction. If there is no immediate predicate producer (not guarded: all FSMs in the row are in the Init and/or CI), pred0 is assigned to signify unconditional execution.

4. The final helper thread instructions, the live-in register sets, and the start PC of the loop are written into the HTC. At the end of the epoch, only the instructions that are selected to be included in the helper thread cache are retrieved from the HTCB, and the entire HTCB is flushed to set up for the next loop construction.

The HTC, in our experiments, can hold the helper threads for up to four loops. It is organized into four rows. Each row is dedicated to a loop and holds up to 128 helper thread instructions. The loop is tagged with its start PC (the target of the outermost loop branch). A flag indicates whether the loop is a nested loop or a non-nested loop. If nested, the row is divided into two halves, the outer-thread's instructions are packed into the first half and the inner-thread's instructions are packed into the second half.

The helper thread instructions include the original instruction obtained from the HTCB and also the additionally encoded predicate source operand. Even though only stores and predicate producers really require this additional source operand, every instruction in the helper thread is assigned the additional predicate source operand only to maintain the fixed-length encoding of the RISCV ISA instructions.

Helper thread instruction fetching is purely sequential until the last instruction (loop

branch) is reached, which resets sequencing back to the first instruction. Additional metadata that is stored in the HTC include (1) the locations (instruction offsets in the row) of up to two loop branches (one for inner-thread-only or outer-thread, one for inner-thread), (2) the location of the inner loop's header branch within outer-thread, and (3) up to three live-in register sets (one for inner-thread-only or outer-thread, two for inner-thread).

5.2.6 Triggering the helper threads

As the main thread retires instructions, their PCs are compared against the start PCs of loops in the HTC. If there is a hit:

1. The pipeline is squashed (no in-flight instructions).

2. The superscalar core's frontend stages and FIFO structures (Reorder Buffer (ROB), Physical Register File (PRF) free list, Load Queue (LQ), and Store Queue (SQ)) are partitioned for a complexity-effective design and isolated processing of the main thread and helper thread. The Scheduler/Issue Queue (IQ) and execution lanes are flexibly shared. Table 5.1 shows the fractional allocation for the threads' partitions, applied to both frontend pipeline stage width and resources.

3. The fetch stage partition of each helper thread injects annotated move instructions, one per logical register in its live-in register set with respect to the main thread. If the helper thread is inner-thread-only or outer-thread, it immediately starts fetching instructions from its HTC entry after injecting the moves (if inner-thread, it waits). When an annotated move instruction reaches the helper thread's rename stage partition, its source is renamed to the corresponding mapping in the main thread's Rename Map Table (RMT). Its destination is renamed as any destination in the helper thread would be: pop a free register from the helper thread's free list partition and update the helper thread's RMT with the new mapping. Thus, when the move executes, it copies a live-in value from the main thread to the helper thread.

4. Instruction fetching in the main thread only resumes when the last of the annotated moves retires. Though not optimal, this is a simple way to ensure that each helper thread obtained its live-in values before the main thread frees the physical registers from which the values were obtained.

Table 5.1: Fractional allocation of frontend width and resources, among main thread (MT), inner-thread-only (ITO), outer-thread (OT), and inner-thread (IT).

active threads	MT's partition	OT's partition	IT's partition
MT + ITO	$\frac{1}{2}$	n/a	$\frac{1}{2}$
MT + OT + IT	$\frac{1}{2}$	$\frac{1}{8}$	$\frac{3}{8}$

5.2.7 Terminating the helper threads

Pre-execution is terminated when the inner-thread-only or outer-thread loop branch resolves as not-taken, exiting the region of interest. Because non-delinquent loop side-exit branches are not included in the helper threads, the main thread terminates pre-execution when it retires an instruction whose PC is outside the PC bounds of the loop being pre-executed. An exception recovery in the main thread also terminates the helper thread. After terminating the helper threads, the pipeline is squashed and all resources are returned to the main thread.

5.2.8 Modifications to the core

This section details the new hardware structures required for implementing Phelps along with other changes required to the superscalar core. The storage overhead of all these structures is listed in Table 5.2. Figure 5.6 pictorially summarizes these hardware structures along with the partitioning scheme (for the inner-thread-only configuration). The structures shown in the figure with the dashed outlines are the newer structures needed for Phelps.

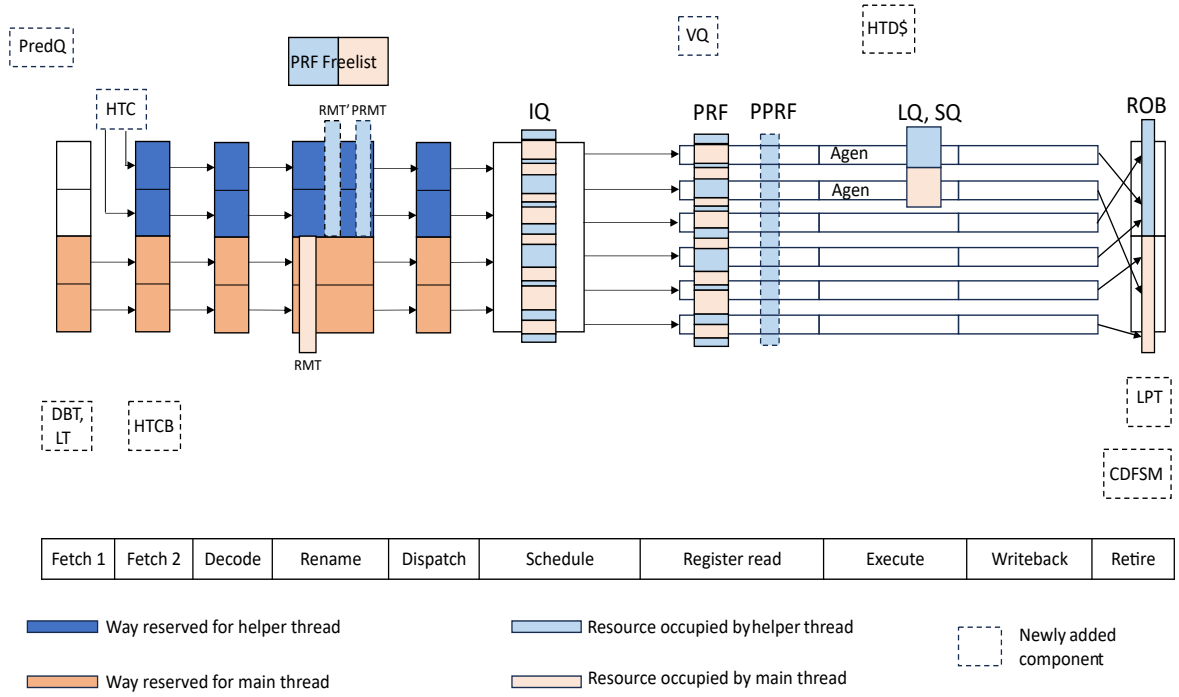


Figure 5.6: New hardware components needed for Phelps along with a visualization of the core's partitioning scheme.

Changes to the core

- Two extra RMTs to support up to two active helper threads.
- The scheduler accommodates an extra source tag in its wakeup CAM, for the extra source operand for stores and predicate producers.
- Partitioning support for the core's frontend stages and FIFO structures for a complexity-effective design and isolated processing of the main thread and helper thread. The FIFO structures include Reorder Buffer (ROB), Physical Register File (PRF) free list, Load Queue (LQ), and Store Queue (SQ). The Scheduler/Issue Queue (IQ) and execution lanes are flexibly shared.

Table 5.2: Storage overhead of Phelps.

Component	**Parameters**	**Cost**
Components for Helper Thread Construction		
Delinq. Branch Table (DBT)	256 entries, fully-assoc.	5,280 B
DBT-Max	32 entries, fully-assoc.	84 B
Loop Table (LT)	8 entries, fully-assoc.	170 B
Helper Thread Construction Buffer (HTCB)	256 inst., 4B/inst. metadata	1,024 B 62.125 B
Last Producer Table (LPT)	32 entries, 30 bits/entry	120 B
queue to detect needed stores	16 entries, 94 bits/entry	188 B
CDFSM matrix, branch list, PC-to-row conversion table	32 rows x 16 col. x 2 bits 16 entries, 5 bits/entry 32 entries, 35 bits/entry	128 B 10 B 140 B
Components for Helper Thread Execution		
Helper Thread Cache (HTC)	4 x 128 inst x 38 bits/inst 4 x 180 bits metadata	2,432 B 90 B
Visit Queue	16 visits, 4 live-ins/visit, 70 bits/live-in	560 B
Prediction Queues	16 queues, 32 iterations 16 PC tags	64 B 60 B
speculative D$ for HT stores	16 sets, 2 ways, 8B block metadata	256 B 236 B
pred-PRF, pred-FL, 2 pred-RMTs	128 reg., 2 bits/reg. 97 entries, 7 bits/entry 2x 31 entries, 7 bits/entry	32 B 84.875 B 54.25 B
Total Cost		**10.82 KB**

Prediction queues

As introduced earlier, Phelps uses 32-entry per-PC prediction queue for communicating pre-execution outcomes from the helper thread to the main thread. All the prediction queues allocated to a single thread are operated in a lockstep manner and thus they all share the same head, spec-head, and tail pointer. There are two such pointers to support pre-execution from doubly nested loops, as both the helper threads are fully decoupled and processing different iterations of the loop.

Each entry in the prediction queue contains a single bit that contains if the corresponding prediction was taken (1) or not-taken (0). There are 16 such queues to support pre-execution of 16 different delinquent branches. The prediction queues all store the PC of the delinquent branch that they are allocated to as tags. As the main thread's fetch unit is fetching with pre-execution enabled, conditional branch PCs in the bundle are looked for a match with the PC tags of all the prediction queues. A hit in one of the prediction queues, reads out the prediction from the entry pointed to by its spec-head pointer and overrides the core's default prediction.

Visit queue

One of the key factors that helps Phelps provide timely predictions is its ability to create and deploy, small and decoupled helper threads. The decoupling of the helper threads enables the outer thread to run far ahead without having to wait for the inner thread to catch up. The outer thread running ahead can extract memory-level parallelism by initiating cache misses from future iterations of the outer thread far ahead of their demand by the inner thread as well as the main thread. While the outer thread runs independently of the inner thread, it may produce live-in values that are required for triggering the inner thread in the near future. These values are communicated from the outer thread to the inner thread through a structure called visit queue. The visit queue is a FIFO circular queue of tables that is used to communicate values from the outer thread to the inner thread. Each of the tables

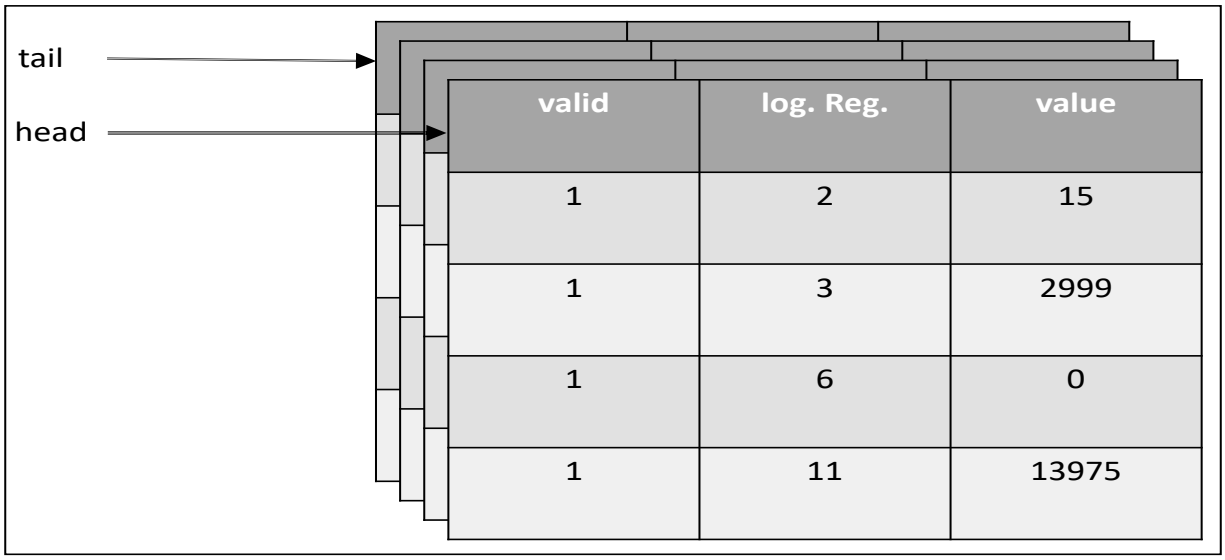


Figure 5.7: Visit Queue for outer-thread to queue inner loop visits and their live-in values for processing by inner-thread.

contains live-in values produced by one iteration of the outer thread along with their register specifiers. Figure 5.7 shows the structure of the visit queue discussed above. The visit queue has three pointers namely, the head, the spec head, and the tail pointers. The outer thread is responsible for manipulating the visit queue's tail pointer and the entries are populated at the tail pointer. Likewise, the inner thread manipulates the spec head and the head pointers and pops entries from the visit queue.

For the case of two helper threads, inner-thread is idle until outer-thread queues a visit to the inner loop. Visits are queued in the Visit Queue depicted in Figure 5.7. When outer-thread retires a not-taken instance of the inner loop's header branch, it allocates a new Visit Queue entry at the tail and writes live-in values (for the inner-thread's second live-in register set supplied by outer-thread) for that visit at slots in the tail entry. When the inner-thread determines that its current inner loop visit has fully iterated–*i.e.* when its loop branch resolves as not-taken –it dequeues the next visit from the head entry. Inner-thread injects move instructions to copy live-in values from the Visit Queue. Each move's source is renamed to the correct slot in the head entry of the Visit Queue, and its destination is

renamed to a free physical register obtained from inner-thread's free list partition. Since outer-thread places values in the Visit Queue, it need not wait for inner-thread's moves to read the values.

Predicate register file

Logical predicate source/destination operands are re-named to physical predicate registers. Thus, we add a 128-entry predicate physical register file (pred-PRF), predicate physical register free list (pred-FL), and two predicate rename map tables (pred-RMT); when two helper threads are running, pred-FL is partitioned 1/2 for outer-thread and 1/2 for inner-thread. Each predicate register is only 2 bits wide: the most-significant bit (msb) indicates whether the predicate producer was predicated-true (enabled) or predicated-false (suppressed) by its predicate producer, and the least-significant bit (lsb) indicates the taken/not-taken outcome of the predicate producer. The consumer of a predicate register is predicated-true if: ((msb == 1) && (lsb == enabling direction of consumer)).

Helper thread's private data cache

Helper thread stores that are predicated true are committed from its SQ partition to a small private cache (32 doublewords organized in 16 sets, 2-way set-associative). Data evicted from this cache is simply lost. Note that this is not an issue as this mini cache is only used by the helper threads and they are allowed to be inaccurate as they would only result in divergence from the main thread. If a helper thread load re-references the same address, it may get stale or up-to-date data, depending on if the main thread's counterpart store already reproduced the data in its L1 cache [PSR02].

5.2.9 Omitting More Complex Scenarios

In our research, we explored and devised methods to support more complex scenarios in Phelps. However, they add significant complexity with no more profitable results. We note,

however, that these features are possible.

Non-delinquent control flow in the backward slice:

We explored including non-delinquent unbiased branches in the helper thread if they guard other instructions that are added to the helper thread. These branches are essentially "affector" branches to a delinquent branch. Phelps does have a way of including affector branches in the backward slice of the delinquent branch.

To implement this, the CDFSM matrix needs to allocate rows for all instructions in the loop. Columns of the CDFSM matrix should be allocated for all branches in the loop. In the process of growing the helper thread, Phelps needs to pause multiple times to add additional branches as seeds to the helper thread: those that were not among the original seeds, are unbiased and guard any instruction in the helper thread thus far. These branches are left as control-flow in the helper thread if predictable with a bimodal predictor, or converted to predicate producers otherwise. Either way, they must be predicate consumers. Including these branches in the helper thread has the following effects:

- There is a significant increase in the implementation cost and complexity.
- Bigger helper threads are formed due to the additional seeds' backward slices being included.
- More instructions in the helper thread might have to be predicated as these affector branches could be brought in as predicate producers if they are unpredictable by bimodal.

These factors do not justify the minimal improvement in performance it provides.

Alternate producers:

An instruction in the helper thread, that is control-independent of a prior branch, may depend on different producers of its source register depending on the direction of the branch.

These producers may be referred to as alternate producers. The presence of alternate producers can be detected by recording the last producer PC alongside each source of each instruction added to the helper thread. If during the helper thread construction, in a different iteration of the loop, a different PC is observed as the producer from the LPT, both producers are deemed as alternate producers. Both alternate producers are added to the helper thread and are converted to predicate consumers. Whenever predicated-false, these instructions need to be converted to a move of the previous version of their logical destination register.

Complex guarding scenarios:

We explored more complex guarding scenarios, such as those created by if statements with OR expressions. Handling such scenarios requires multiple predicate source operands in the helper thread instructions. Thus each entry in the issue queue would also need an additional source operand to keep track of. The overall predicate evaluation for the predicate consumer involves the ORing of the predicates evaluated from each of its predicate producers. The scenario is detectable as multiple CD states in a row of the CDFSM matrix. This scenario is however not frequently observed in graph-based workloads, so handling them does not justify the added complexity.

5.2.10 Ineligibility of delinquent loops

A loop that was deemed delinquent during the delinquency analysis (first stage) of helper thread construction could still be ineligible for three reasons:

- If the resultant helper thread contains more than 75% of the instructions in the original loop, it is deemed ineligible. Such loops are not profitable for pre-execution by Phelps as the helper thread is not much shorter than the main thread. The lack of reduction in the loop size also ends up just as slow as the main thread. Pre-executing such loops

with Phelps could even lead to performance degradation, as a significant fraction of the resources are taken away from the main thread.

- If the outermost loop is itself a short trip count loop, it is deemed ineligible for pre-execution. Phelps incurs a high overhead for entering and exiting the loop due to the pipeline squashes needed to reconfigure. Pre-executing such short loops using Phelps could incur more penalties due to the frequent squashing of the pipeline. This is more likely to outweigh any performance benefit gained from pre-executing its delinquent branches. So such loops are candidates for ineligibility. Such loops can be detected and made ineligible in the delinquency analysis phase as it requires just tracking how frequently an outer loop is visited in an epoch.

- In the case of outer thread - inner thread pairs, if there is a flow of values from the inner helper thread to the outer helper thread, the loops are not pre-executable. Decoupling such loops is incorrect as the outer thread cannot process another iteration until the inner thread completes its visit. This scenario can be detected in the helper thread construction phase when an instruction included in the outer helper thread references a PC that is within the inner thread's loop bounds.

5.2.11 Delinquent branches nested within a function call

A key challenge faced by Phelps is its inability to detect delinquent branches that are not truly nested within a loop. This could happen in cases where the delinquent branch(es) are present in a function that is iterated a lot. The inability stems from the fact that Phelps' delinquent branch identification phase considers a branch to be nested within the loop only if it lies within the bounds of the loop. This is however not true in the case of branches that exist inside a non-inlined function. So, Phelps fails to include this delinquent branch as part of the loop and only pre-executes other delinquent branches. This scenario was observed in a couple of SimPoints of the *mcf* and *leela* benchmarks in the SPEC2017 benchmark suite.

Chapter 6

Methodology and Results

6.1 Methodology

For our experiments, we use an in-house, RISC-V, execution-driven, cycle-level, execute-at-execute simulator to model the superscalar core and the additional components required for modeling Phelps. The superscalar core and memory hierarchy are shown in Table 6.1. Of note is that state-of-the-art high-performance cores have scaled to large windows and superscalar widths. For example, reverse engineering by AnandTech [Fru20] of Apple's A14 chip inferred an ROB size of around 630 instructions and an LQ size of 148 loads. Accordingly, the superscalar core used in our experiments has a ROB size of 632 and an LQ size of 144 (to be divisible by 8 for partitioning among the main thread and helper thread(s)). We perform our experiments on the SPEC 2006 integer benchmark *astar* with reference input *rivers* (astar_-rivers). We also test the effectiveness of Phelps on graph workloads by simulating benchmarks from the GAP benchmark suite. For all the benchmarks from the GAP suite, we use the roadNet-CA graph as the input dataset. We use the SimPoints methodology [SPHC02] to obtain up to 5 representative regions of 100 million instructions for our simulations and compute the weighted harmonic mean of IPCs over a benchmark's SimPoints to obtain the overall IPC of the benchmark.

Branch Runahead: As already mentioned in Chapter 3, we implemented the core-only version of Branch Runahead [PP21]. As with Phelps' MT+ITO configuration (Table 5.1), the core's frontend pipeline stages, PRF free list, and LQ are partitioned 50/50 with the main thread executing in one partition and all chains executing in the other (the main thread has the whole ROB and SQ to itself).

Table 6.1: Superscalar core and memory hierarchy configuration.

branch predictor	64KB TAGE-SC-L [Sez16]
pipeline depth	11 stages (fetch to retire)
fetch/retire width	8 instr./cycle
issue/execute width	8 instr./cycle
execution lanes	4 simple ALU, 2 load/store, 2 FP/complex ALU
ROB/PRF/IQ/LDQ/STQ	632/696/128/144/144
L1I cache	32KB, 8-way
L1D cache	48KB, 12-way, 3 cycles (1 agen, 2 hit)
L1D prefetcher	next-N-line (N=2)
L2 cache	1.25MB, 20-way, 15 cycles
L3 cache	3 MB, 16-way, 40 cycles
L2/L3 prefetcher	VLDP (5.5 Kb) [SKB+15]
DRAM	100 cycles
branch checkpoints	64, OoO reclamation

6.2 Results

6.2.1 Speedup

Figure 6.1 shows the speedups of perfect branch prediction (perfBP), Phelps, and Branch Runahead with speculative triggering (BR). Phelps-12w and BR-12w, feature a 12-wide core, in which the main thread gets the same frontend width and resources as the baseline (8-wide frontend, ROB/PRF/LQ/SQ = 632/696/144/144) and the helper threads and BR chains get the same frontend width and resources as they would in Phelps and BR respectively. The IQ and 12 execution lanes are still flexibly shared, although there are 4 more lanes.

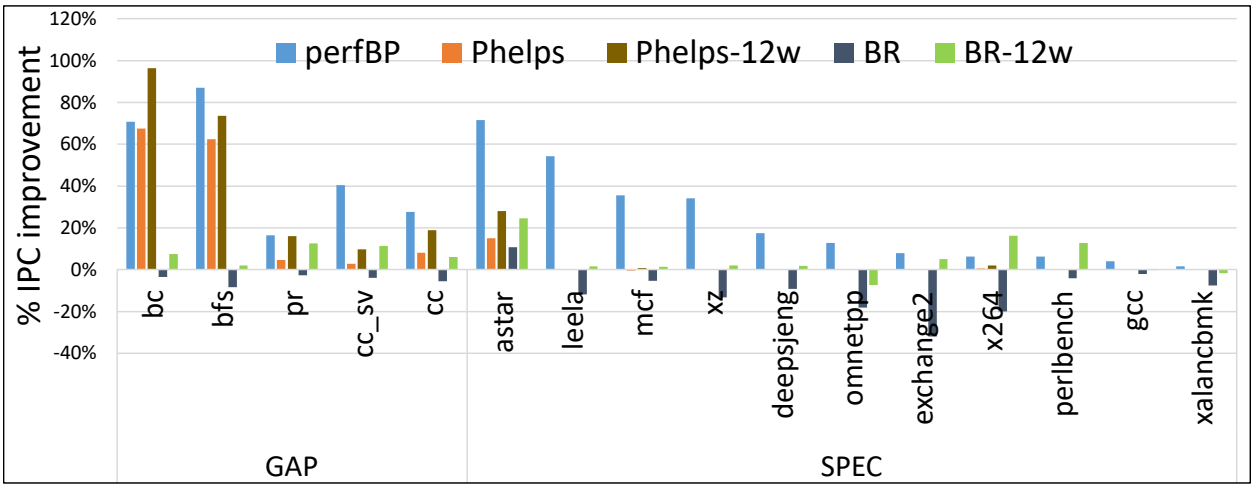


Figure 6.1: Speedup of perfect branch prediction, Phelps, and Branch Runahead.

Phelps yields good speedups on *bc* (68%), *bfs* (62%), and *astar* (15%). *Astar*'s speedup considering all SimPoints is much less than that of the top-weighted SimPoint alone (48%), for two reasons. First, the other SimPoints are not nearly as delinquent. Second, their delinquent loops are not sufficiently long-running per visit to amortize helper thread start/stop overheads, hence they are ineligible.

For GAP benchmarks and *astar*, Phelps-12w shows significant speedups that exceed speedups of Phelps. This is because the main thread in Phelps-12w is not hindered by sharing

the pipeline width and resources with the helper thread. The additional pipeline width and resources combined with good prediction accuracy and a prefetching effect enables Phelps-12w to even exceed the performance of perfect branch prediction in the case of *bc*.

BR exhibits mostly slowdowns, except for *astar*. This is due to (1) the main thread getting only half frontend width, LQ, and PRF for the full run, and (2) inaccurate speculative triggering combined with not enough chain group level parallelism, yielding many untimely predictions. BR-12w turns things around with mostly speedups due to not affecting the main thread by taking away resources from it.

The SPEC benchmarks are sorted in Figure 6.1 from highest to lowest speedup with perfBP. Phelps rarely or never activates helper threads in SPEC 2017. *omnetpp* through *xalancbmk* (6 benchmarks) are not misprediction limited, in any case: perfBP yields only 1%-12% speedup. Although *leela*, *mcf*, *deepsjeng*, and *xz* present more opportunities, Phelps is unable to pre-execute delinquent branches and loops in these benchmarks. To understand why, we perform two experiments. In the first experiment, we categorize the mispredictions in both GAP and SPEC benchmarks based on their sources. This helps us understand why that branch instance was not pre-executed or pre-executed incorrectly. In the second experiment, we try to understand the misprediction density of the benchmarks.

Categorizing the sources of the residual mispredictions

Broadly, all the branch mispredictions incurred can be categorized into mispredictions from Phelps's predictions (incorrect pre-execution) and TAGE-SC-L's predictions. Note that a branch is only predicted by TAGE-SC-L if there is no pre-execution taking place for that static branch. This is because we opt to stall the main thread's fetch stage when a pre-execution outcome is not timely. Mispredictions arising due to the core's default predictor may further be classified based on the delinquency of the branch and the successful construction of a helper thread containing the branch. For this experiment, we use an FSM for each static branch to track its most recent state of delinquency and helper thread construction.

The states of the FSM are listed below:

- gathering delinquency: The branch is in this state as it was allocated a DBT entry in this epoch so we do not have enough information regarding its delinquency yet.
- not delinquent: The branch is deemed not delinquent since it did not clear the delinquency threshold.
- del. but not in loop: The branch clears the delinquency threshold, however, it is not deemed to be in a loop by Phelps' delinquent loop analysis.
- del. but ht being const.: The branch clears the delinquency threshold, however, a helper thread for its loop(s) is being constructed in this epoch.
- del. but ht not const.: The branch clears the delinquency threshold, however, its helper thread is not being constructed in this epoch since a more delinquent loop was picked.
- del. but ot/ito not iterating enough: The branch cleared the delinquency threshold, however its outermost loop was not iterating enough, thereby becoming ineligible.
- del. but ht too big: The branch cleared the delinquency threshold, however, its helper thread was not much shorter than the loop.
- del. but it->ot comm.: The branch cleared the delinquency threshold, however, data flows from its inner thread to the outer thread, thereby deeming ineligible for pre-execution.
- pre-executed but misp.: The branch was pre-executed incorrectly by Phelps due to the influence of alternate producers or missing stores.

At the end of each epoch, the states of every branch that holds a DBT entry are updated to reflect the most recent state based on its delinquency (del. or not del.) and failure condition (ht too big, it->ot comm., ot/ito not iterating enough). For the subsequent epochs, when a

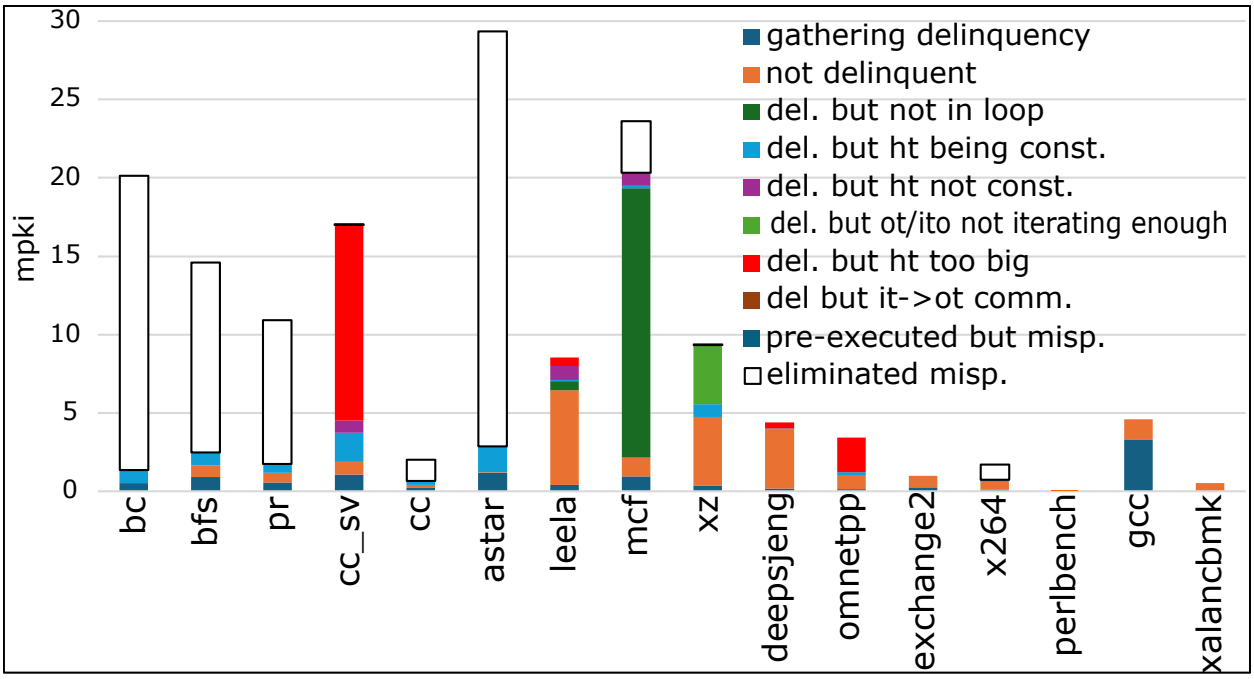


Figure 6.2: Characterizing mispredictions based on their sources

branch misprediction is observed in the main thread's fetch unit, the branch's DBT entry is looked up and the counter corresponding to the state of the branch is incremented. Fig 6.2 provides the breakdown of the sources of the residual mispredictions (in MPKI) when using Phelps. The stacked histograms correspond to the breakdown of mispredictions for the top-weighted SimPoint for all the benchmarks.

In benchmarks *bc*, *bfs*, *pr*, *cc*, and *astar*, Phelps manages to eliminate most of the mispredictions via pre-execution. This is evident from the relatively larger "eliminated misp." block (white block) compared to the colored blocks. The remaining mispredictions are observed to be either from the first stage of the helper thread construction pipeline where we are gathering delinquency information about branches (blue) or from the second stage (light blue) where we are constructing a helper thread for the delinquent branches.

The presence of a purple sliver in *cc_sv* indicates that there is more than one loop being simultaneously detected as delinquent, however Phelps is only able to construct one helper thread at a time and thus suffering mispredictions for longer time on the other loops. Re-

gardless, Phelps is unable to eliminate mispredictions in this Simpoint because, the helper threads being constructed for the delinquent loops end up too big (red) thereby deeming themselves ineligible for pre-execution. Additionally, note that the size of the orange block in all these benchmarks is pretty small indicating that there are only very few branches in these benchmarks that are not clearing the delinquency threshold.

Looking at the SPEC2017 benchmarks, except *mcf*, all other benchmarks are simply not as delinquent as the GAP benchmarks. While branches in *mcf* are delinquent, most of the mispredictions are due to the branch deemed to not be within a loop (dark green). This may happen when the loop contains a function call and the delinquent branch is within the non-inlined function. In such a scenario, the delinquent branch's PC is not within the bounds of the loop and thus fails to be detected in a loop. In benchmarks *leela*, *deepsjeng*, *exchange2*, and *xalancbmk*, the orange bar is quite significant, indicating that much of the MPKI contribution is from branches that are not individually themselves delinquent. Even the few branches that clear the delinquency threshold in *leela*, *deepsjeng*, and *omnetpp* end up constructing a helper thread that is too big. Thus, Phelps is unable to pre-execute branches in these benchmarks. *Xz* also shows a decent concentration of non-delinquent branches, but the ones that clear the delinquency threshold are found to not iterate enough (light green) which renders the helper thread not profitable for pre-execution. *x264* manages to successfully construct a useful helper thread, however the benchmark is not really branch predictor limited. The relatively large "gathering delinquency" component in *gcc* is due to repeated evictions of branches from the DBT due to the benchmark having a big instruction footprint and a large number of static branches.

Understanding misprediction density of benchmarks in GAP and SPEC2017

For the second experiment mentioned above, we study the nature of mispredictions in the SPEC2017 and GAP benchmark suites to understand if the overall MPKI of the SimPoint is contributed to by a few highly delinquent branches or by many lesser delinquent branches.

This helps us to identify benchmarks with high misprediction density, i.e. benchmarks in which most of the MPKI is contributed by just a few highly delinquent branches. Such a trend is indicative of the presence of long-running loops with a high concentration of mispredictions, which are good candidates for achieving speedups with Phelps.

For this study, we simulate 100 million instructions of all SimPoints in each benchmark suite using the baseline superscalar core. For each such run, we sort all the observed conditional branches in the decreasing order of their contribution to the overall number of mispredictions. We then plot the cumulative MPKI (Y-axis) against the number of conditional branches (involved in the accumulation) up to a maximum of 30 static branches (X-axis). The resulting plot is a monotonically increasing curve that tends towards the overall MPKI of the run.

Figure 6.3 depicts the cumulative MPKI curve for the GAP benchmarks *bc*, *bfs*, *pr* and *cc_sv*. From the graphs, it can be seen that for *bc*, *bfs*, and *cc_sv* the majority of the overall MPKI is due to a very few number of delinquent branches. The graphs for these benchmarks flatten out quickly indicating that there are not many more delinquent branches. In the case of pr, it is just one delinquent branch that causes almost all the mispredictions. This can be inferred from the flat-line graph that starts at a high value.

Figure 6.4 depicts the cumulative MPKI curve for the GAP benchmark *cc*, SPEC2006 benchmark *astar_rivers*, and SPEC2017 benchmarks *mcf* and *leela*. From the graphs, it can be seen that for *cc* one of the SimPoints shows a very slowly increasing curve near 0. This indicates that this SimPoint contains many conditional branches that are not very delinquent. The other two SimPoints for the same benchmark show steep slopes followed by a flat line indicating that the most mispredictions in these SimPoints are due to a very few number of branches.

The *astar* benchmark from SPEC2006 shows that only one of its SimPoint (actually the top-weighted SimPoint) has delinquent branches. In fact, there are 16 of them as we studied earlier in Chapter 2. A few of the SimPoints from *mcf* show a few very bad branches and the

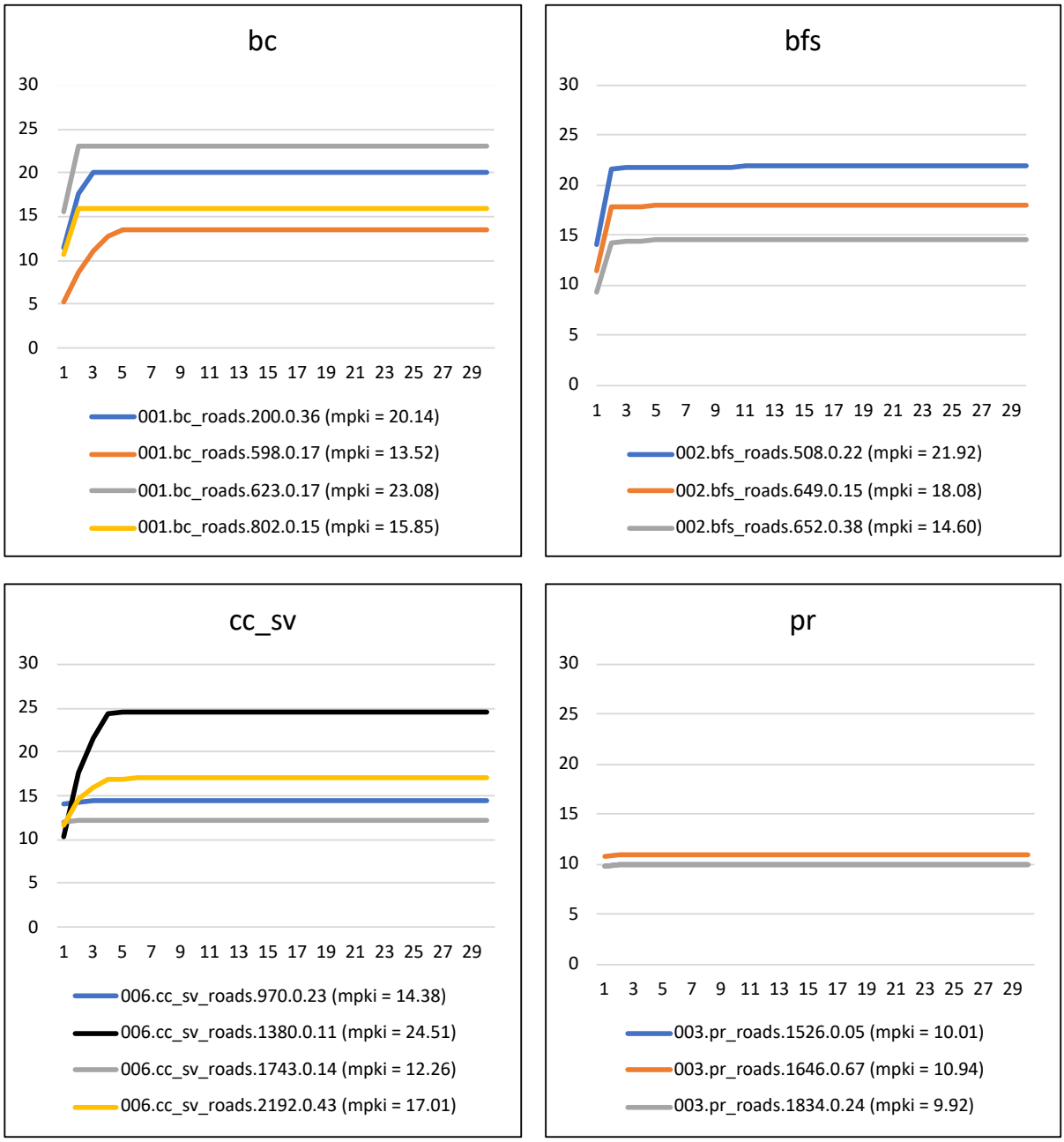


Figure 6.3: Cumulative MPKI contribution (Y-axis) is plotted against the top 30 static branches of each SimPoint sorted by their MPKI contribution (X-axis). Different curves are used for different SimPoints, whose weights and overall MPKI are indicated in the legend. Benchmarks covered: *bc*, *bfs*, *cc_sv*, and *pr*

curve starts to taper out. Phelps would be expected to do well in the GAP benchmarks and *astar* and *mcf*, due to them having highly delinquent branches. *Leela*, on the other hand, has

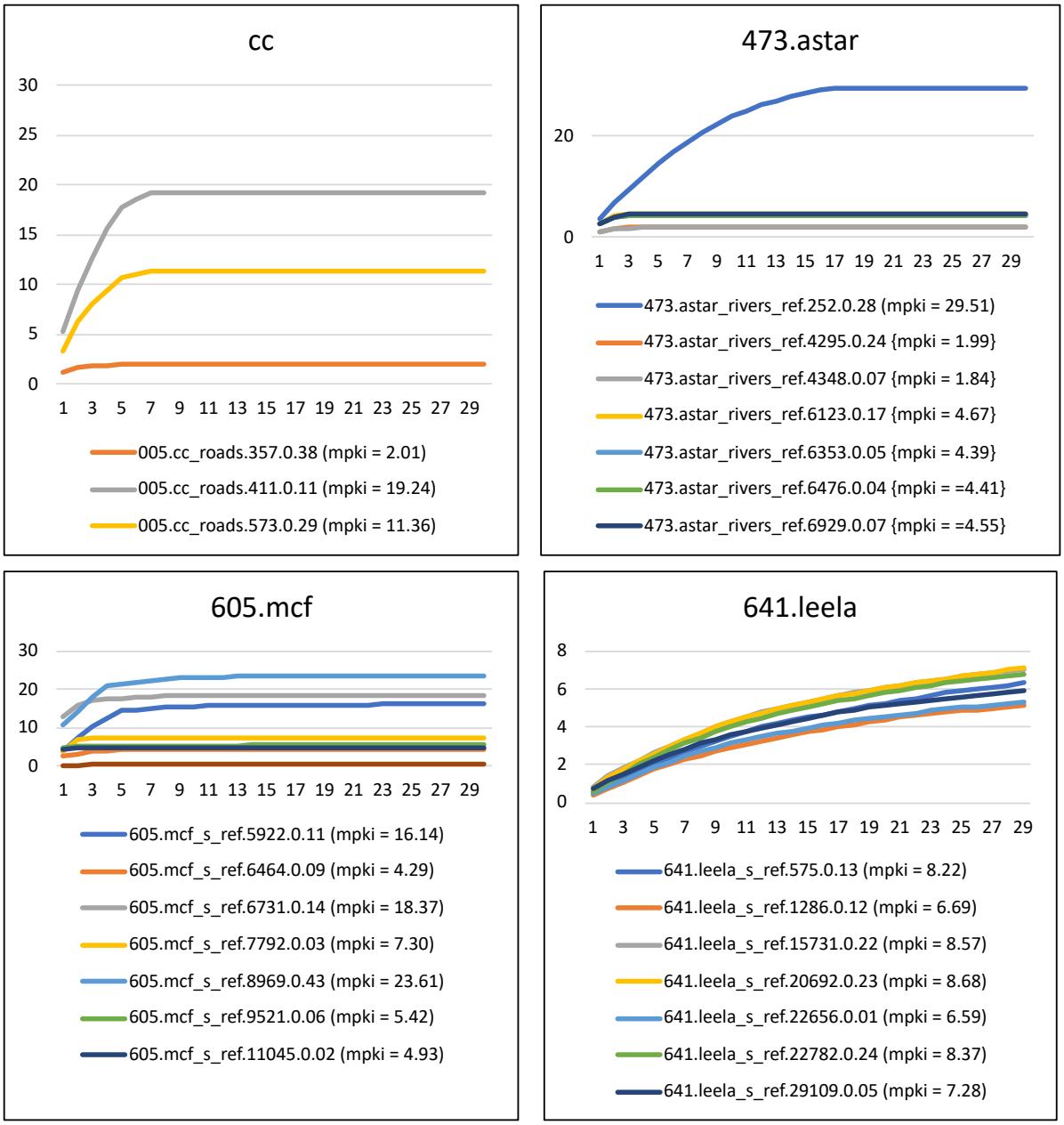


Figure 6.4: Cumulative MPKI contribution (Y-axis) is plotted against the top 30 static branches of each SimPoint sorted by their MPKI contribution (X-axis). Different curves are used for different SimPoints, whose weights and overall MPKI are indicated in the legend. Benchmarks covered: *cc*, *astar*, *mcf*, and *leela*

a very slowly increasing curve and thus has a lot of branches contributing towards its overall MPKI. However, most of these branches are not individually delinquent as they don't clear the required MPKI threshold of 0.5.

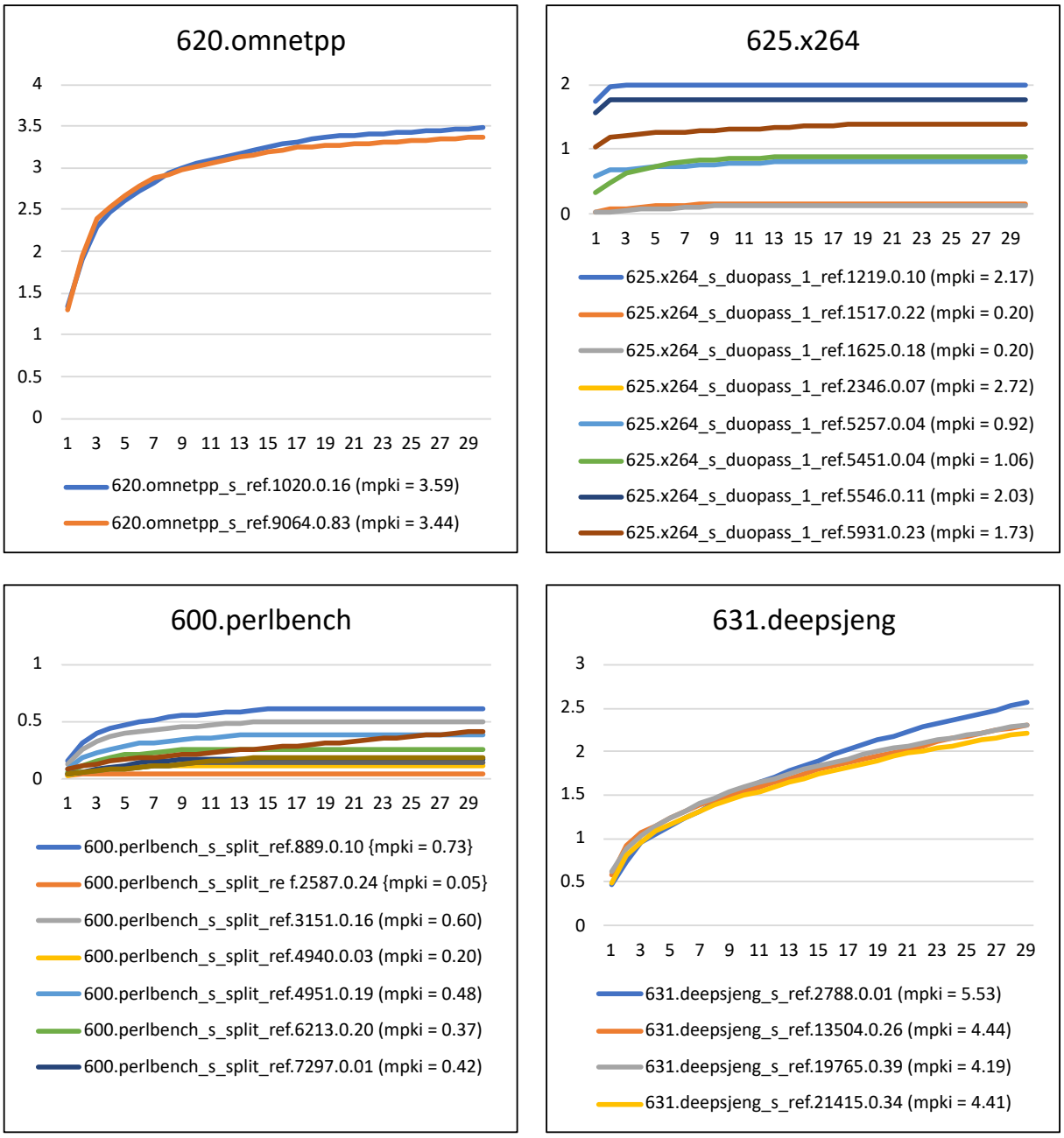


Figure 6.5: Cumulative MPKI contribution (Y-axis) is plotted against the top 30 static branches of each SimPoint sorted by their MPKI contribution (X-axis). Different curves are used for different SimPoints, whose weights and overall MPKI are indicated in the legend. Benchmarks covered: *omnetpp*, *x264*, *perlbench*, and *deepsjeng*

Figure 6.5 depicts the cumulative MPKI curve for the SPEC2017 benchmarks *omnetpp*, *x264*, *perlbench* and *deepsjeng*. All of these benchmarks show relatively lower overall MPKI

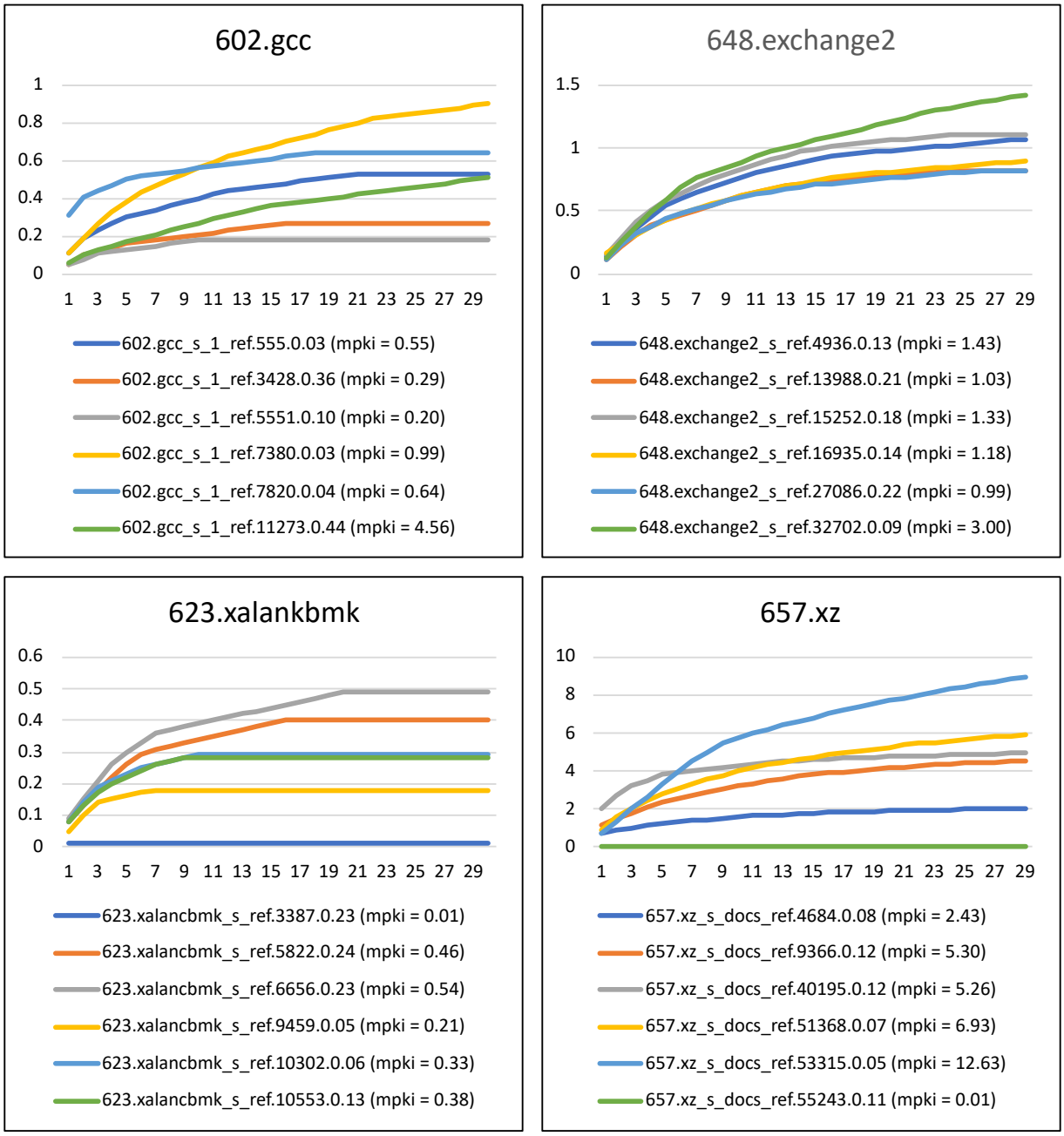


Figure 6.6: Cumulative MPKI contribution (Y-axis) is plotted against the top 30 static branches of each SimPoint sorted by their MPKI contribution (X-axis). Different curves are used for different SimPoints, whose weights and overall MPKI are indicated in the legend. Benchmarks covered: *gcc*, *exchange2*, *xalancbmk*, and *xz*

than the benchmarks we have seen so far. Moreover, Only the top mispredicted branch in some SimPoints of *omnetpp*, *x264*, and *deepsjeng* barely manages to cross the threshold for

delinquency. *Perlbench* has a very low MPKI and even its top mispredicted branch does not clear the delinquency threshold. So Phelps would not be actively pre-executing any branches in this benchmark. This trend is also observed in the *gcc*, *exchange2*, and *xalankbmk* benchmarks shown in Figure 6.6. Only *xz* benchmark from that figure seems to have delinquent branches that clear Phelps' delinquency threshold.

It can be inferred from the study that the GAP benchmarks are likely to have a pretty high misprediction density due to most of their overall MPKI resulting from just a couple of delinquent branches. Astar's top-weighted simpoint, on the other hand, has a lot of contributors to its high misprediction count. However, all of these appear to be contained in a single delinquent loop. From the SPEC2017 benchmark suite, only *mcf* and *xz* appear to have pretty highly delinquent branches in some of their SimPoints. The other benchmarks have relatively lesser overall MPKI and only one or two branches from some of their benchmarks barely manage to clear the delinquency threshold.

In benchmarks with high misprediction density (most of the mispredictions contributed to by just a few branches), the baseline superscalar core would be unable to form a useful instruction window due to frequent mispredictions. Phelps is thus expected to be better suited for pre-executing such benchmarks since the overhead of taking away resources from the helper thread is reduced.

Phelps works well for long-running tight loops with high concentrations of mispredictions. This doesn't mean Phelps cannot support workloads with many different loops, rather, each such loop should be long-running with high misprediction density.

6.2.2 Understanding the speedups attained by Phelps

The speedup attained by any branch pre-execution technique stems from two sources. Firstly, the main thread benefits from streamlined execution due to highly accurate branch predictions obtained through branch pre-execution. Secondly, there is also an important prefetching side-effect due to pre-execution. This is because, the shorter helper thread runs ahead of the

main thread and generates load requests to addresses that would be referenced by the main thread in the near future. By being shorter, the helper thread could expose more memory level parallelism than the main thread could do in isolation. In this subsection, we would like to isolate both these sources to understand if the speedups shown in the previous subsection are mostly due to the reduced mispredictions or due to the prefetching effect.

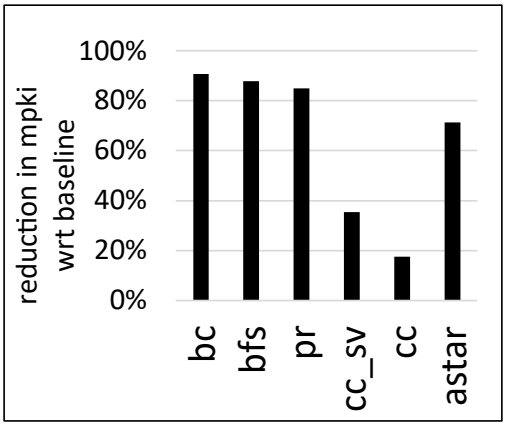


Figure 6.7: Reduction in MPKI due to Phelps.

To examine the first source, we look at the reduction in mispredictions when using Phelps. Figure 6.7 shows the reduction of MPKI across the GAP benchmark suite and the *astar* benchmark when using Phelps. It can be seen that Phelps achieves significant reductions in MPKI (ranging between 71%-91%) on four of the six benchmarks shown. The next three paragraphs explain why this is the case.

Figure 6.8 shows the number of unique loops that were successfully pre-executed in the GAP and *astar* benchmarks. Figure 6.9 shows the number of epochs in which the helper threads were active and pre-executing delinquent branches. Recall that each epoch in our Phelps design is equal to 4 million instructions retired in the main thread. So a run of 100 million instructions has 25 epochs in total. However, since Phelps helper thread construction takes 2 epochs, the maximum number of epochs in which Phelps can pre-execute is 23. This is in fact the case for two *bc* SimPoints, one *bfs* SimPoint, and one *astar* SimPoint.

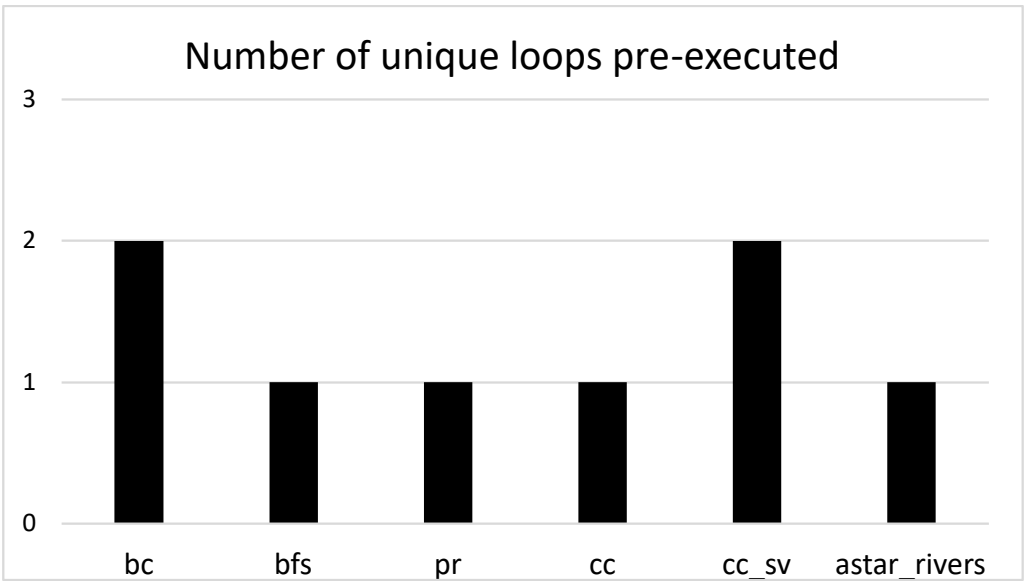


Figure 6.8: Number of unique loops pre-executed by Phelps

From Figure 6.9, it can also be observed that there was no pre-execution ongoing in one SimPoint of *cc_sv* and all but one SimPoint of *astar*. For *cc_sv*, this is due to the fact that the helper thread construction failed in this SimPoint across all epochs due to ineligibility criteria. For the *astar* SimPoints, loops were rendered ineligible due to the fact that the outer loop had a short trip-count and thus pre-execution using Phelps is not profitable in this case.

All other SimPoints show significant pre-execution activity in most of the epochs. This flurry of pre-execution activity combined with a near-perfect pre-execution accuracy results in the significant MPKI reduction seen in Figure 6.7.

To isolate the effect of branch prediction from the effect of prefetching, we perform an experiment wherein we keep all aspects of Phelps unchanged except that we don't push prediction outcomes onto the prediction queues. In this mode, the helper threads are constructed, triggered, and executed as usual. However, the main thread now only uses the predictions produced by the core's default branch predictor. Doing so, simply converts the helper thread into a highly accurate prefetching thread. The difference in performance of this mode when compared against the performance of Phelps can be attributed to the benefit of

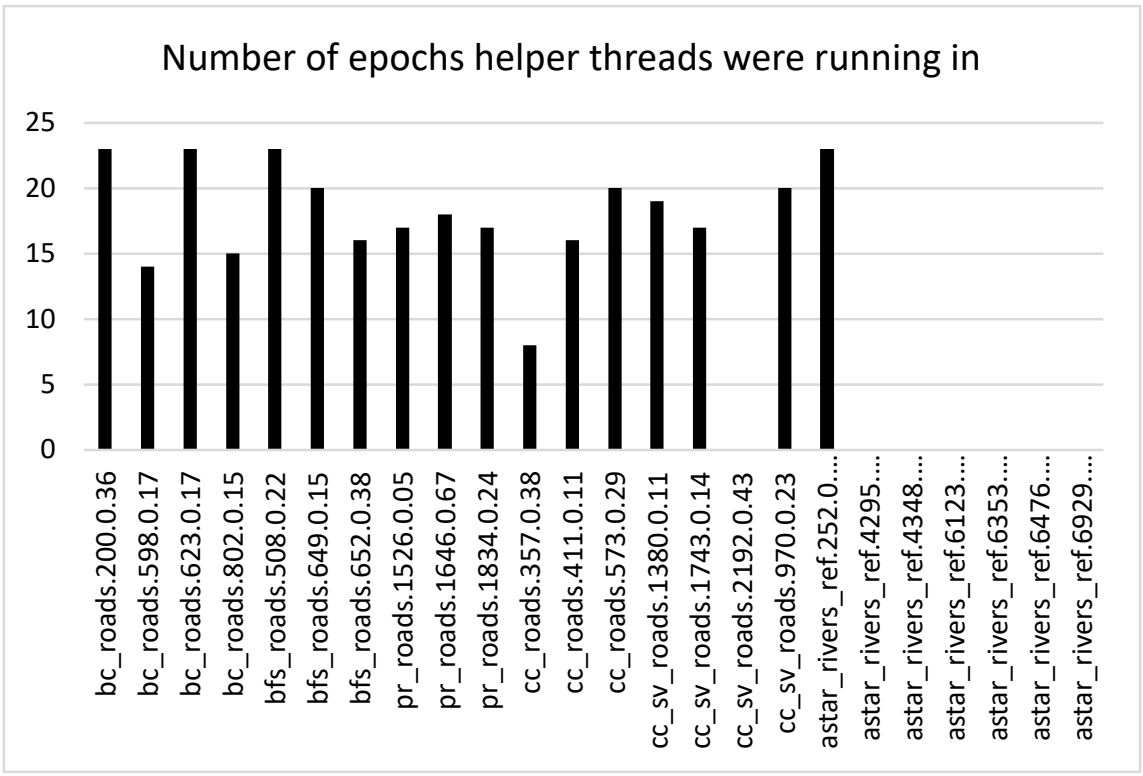


Figure 6.9: Number of epochs in which the helper threads were active

good prediction accuracy.

Figure 6.10 shows the speedup of phelps and phelps-only-prefetch over the baseline. *Bc* achieves a modest speedup of 23% when running Phelps as a prefetching thread. When combined with the good prediction accuracy of Phelps, the speedup jumps to 68%. *Bfs*, on the other hand, benefits more (37%) when using Phelps as a prefetch-only helper thread. Nonetheless, the benefit from good prediction accuracy is still quite significant, as the speedup of Phelps for this benchmark stands at 62%.

In all other benchmarks, using Phelps as a prefetch-only helper thread yields very low speedups and also slowdowns in the case of *astar* and *cc_sv*. The loads performed by the helper threads in these benchmarks are not really delinquent loads and yield very little benefit. Moreover, the cost accrued by taking away half of the resources from the main thread is high and this results in slowdowns over the baseline. So the speedups observed in these benchmarks can be entirely attributed to lower branch MPKI.

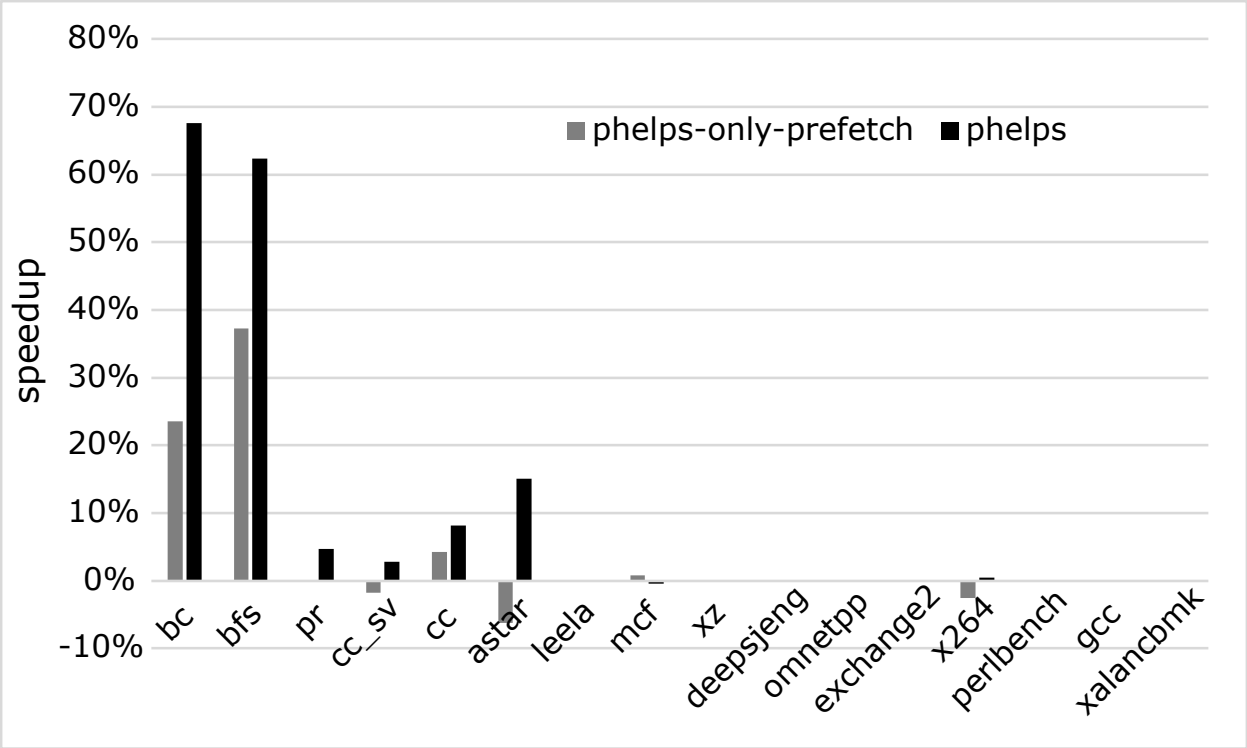


Figure 6.10: Speedup comparison between phelps and phelps-only-prefetch

Similar to other branch pre-execution techniques, Phelps achieves speedups at the cost of partially redundant execution. The overhead, in terms of the number of helper thread instructions retired, is shown in Figure 6.11. However, through precise helper thread construction and by thresholding the sizes of the helper threads, Phelps incurs a mean overhead of 34.7 million helper thread instructions for 100 million instructions retired in the main thread. One must also acknowledge that the speedup stems from reducing branch mispredictions, which in turn reduces the number of wrong-path instructions fetched and executed, and then squashed, in the main thread.

6.2.3 Understanding the impact of including stores in the helper thread

Figure 6.12 examines the importance of including stores in the helper thread. For this study, we compare the performance of Phelps when stores are included and not included in the

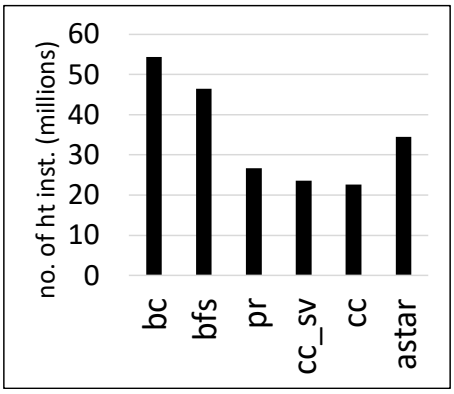


Figure 6.11: Retired helper thread instructions.

helper thread. From the figure it is quite evident that stores are critical for Phelps to achieve its potential on *bc* and *astar*. Both these benchmarks feature stores that 1) influence the outcome of delinquent branches in the loop; 2) are themselves control-dependent on delinquent branches. Including the store instruction within the helper thread and predicating it helps Phelps achieve near perfect pre-execution accuracy on these benchmarks. Figure 6.13 shows the reduction in MPKI when the store is included or ignored in the helper thread. The performance drop in Figure 6.12 can be correlated with the drop in pre-execution accuracy when the store is not included in the helper thread.

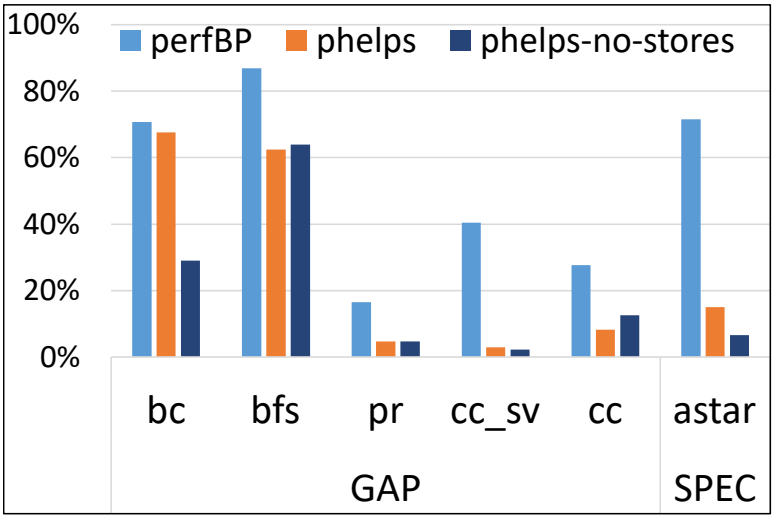


Figure 6.12: Speedup of Phelps with/without stores.

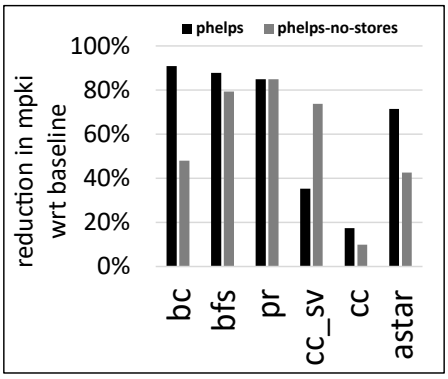


Figure 6.13: MPKI reduction of Phelps with/without stores.

Bfs also has one such store, but the store-load distance in the inner-thread is long enough in most cases that the main thread retires its corresponding store before the inner-thread references the data. Figure 6.13 shows that not including the store in the helper thread does degrade the pre-execution accuracy by a bit. Despite the drop in pre-execution accuracy, benchmarks like *bfs* and *cc* see an increase in the performance. This is because, the lower accuracy is outweighed by more timeliness owing to fewer helper thread instructions.

cc_sv sees an increase in the MPKI reduction when stores are not included in the helper thread. This is because, not including the store in the helper thread made a loop eligible for pre-execution by Phelps. When this loop tries to include the store and its backward slice into the helper thread, it exceeds the helper thread size reduction threshold thereby deeming it ineligible for pre-execution.

6.2.4 Understanding the impact of stalling the main thread when pre-execution is late

In our Phelps design, we had opted to stall the main thread's fetch unit when a delinquent branch (that is being pre-executed by Phelps) is fetched and the prediction queue is empty. To understand its benefit, we also perform an experiment where we let the main thread consume the predictions produced by the core's default predictor in instances where the

pre-executed outcome is not yet available.

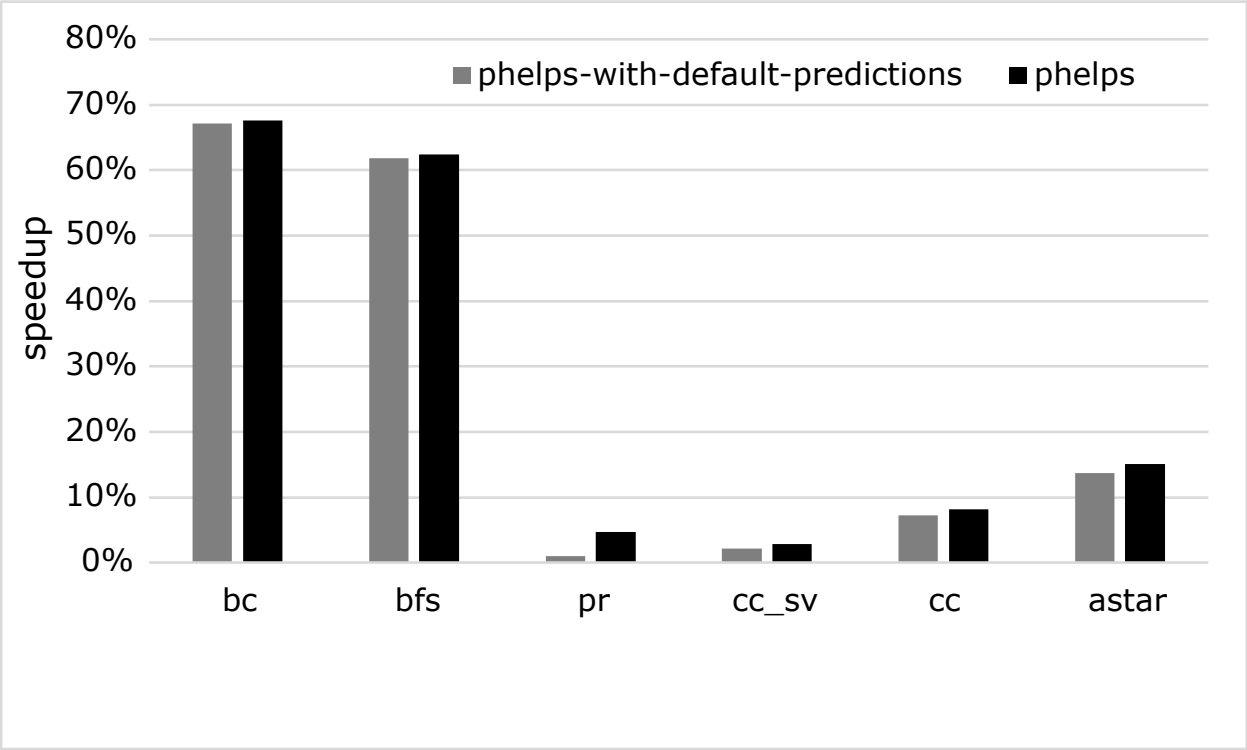


Figure 6.14: Comparing speedups of main thread's fetch policies when pre-execution is late.

Figure 6.14 shows the speedup of both the design policies normalized over the baseline's performance. It is evident that in all the benchmarks shown, using the default predictions whenever Phelps is late leads to a slight dip in performance. Due to the highly delinquent nature of the branches in these benchmarks, using default predictions when Phelps is late, could result in branch mispredictions which ends up degrading performance. However, the fact that the drop in performance is not huge is an indicator that pre-execution by Phelps is very timely. When the predictions are timely, the reliance on the default branch predictor's predictions are minimized thereby leading to good prediction accuracy,

6.2.5 Understanding the impact of partitioning resources among the threads

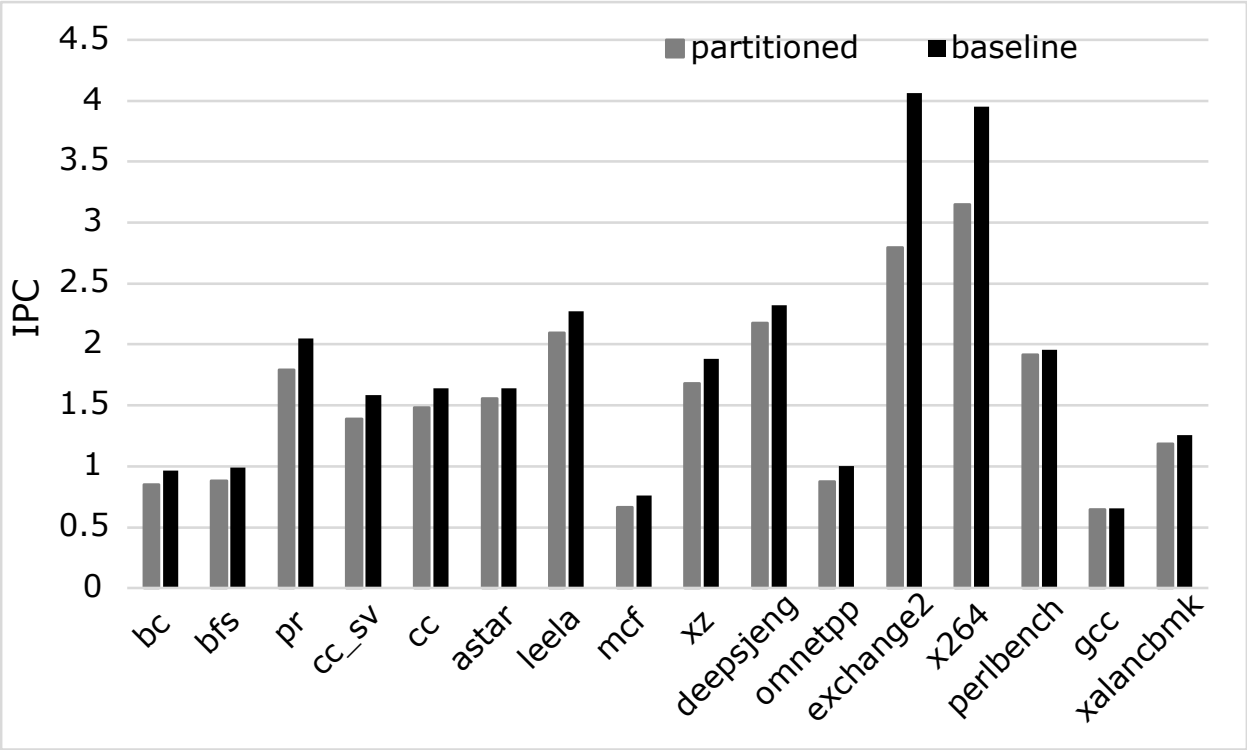


Figure 6.15: Impact of partitioning the resources among the main thread and the helper threads.

A key cost incurred by Phelps is the partitioning of the core's resources among the main thread and the helper thread. To understand the impact of taking resources away from the main thread, we compare the performance of baseline against a processor configuration that is sized similar to the resources available to the main thread when in Phelps mode.

Figure 6.15 denotes the IPC of the partitioned configuration and the baseline configuration for all GAP, SPEC2017 benchmarks, and astar. It can be observed that taking away resources from the main thread could be quite significant. In the case of GAP benchmarks, taking away half the resources from the main thread results in a significant performance reduction in the range of 10%-13%. The effect on some of the SPEC2017 benchmarks is

observed to be quite detrimental. *exchange2* and *x264*, for instance, incur a slowdown of 31% and 20%. This study explains the importance of having the misprediction threshold for determining the delinquency of a branch. By having a lower threshold, Phelps might be attempting to pre-execute branches that might not yield much benefit while simultaneously suffering from the negative impact of taking away resources from the main thread.

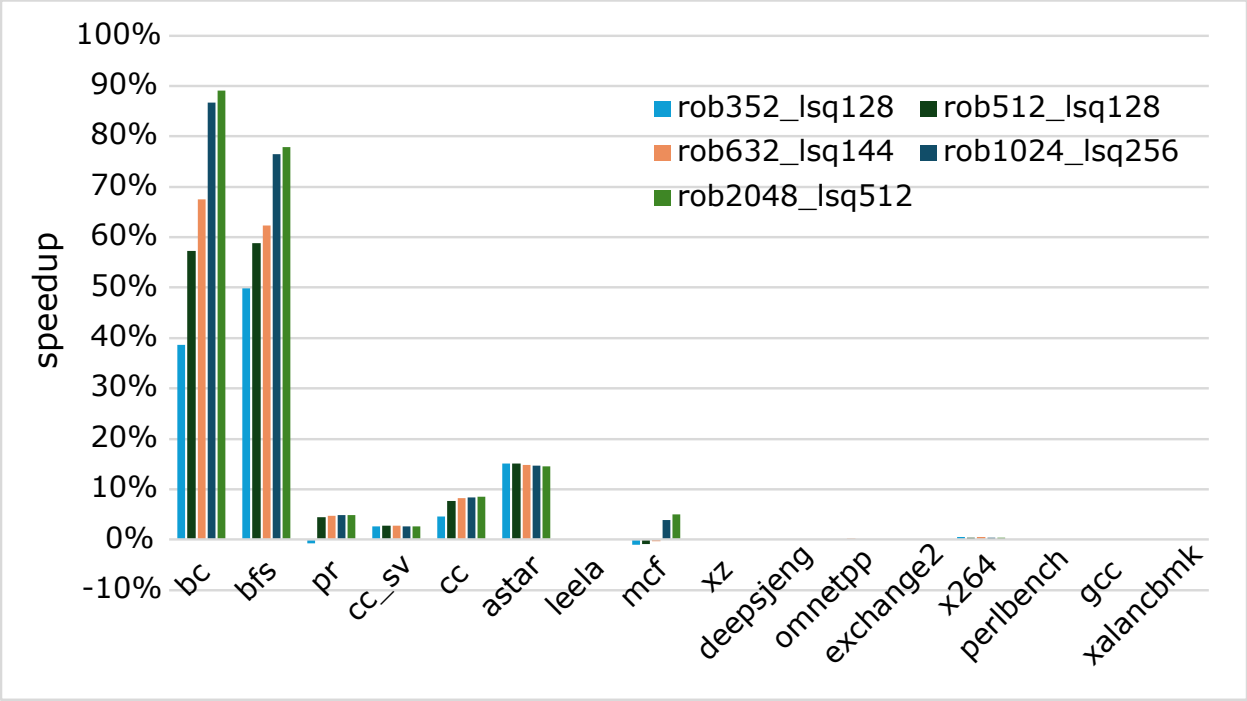


Figure 6.16: Sensitivity of Phelps to instruction window sizes.

We also perform a sensitivity study by sweeping the window sizes for the baseline and Phelps. Figure 6.16 shows the speedup of different window size configurations denoted as robX_lsqY (where X and Y are ROB and LSQ sizes respectively). We observe that for GAP benchmarks like *bc* and *bfs*, increasing the window size further shows significant speedups over a similarly-sized baseline. This is because, the baseline's main thread has the entire resources to itself and increasing the window size beyond a certain saturating point would provide no benefit. However, in Phelps, the helper threads which only get to use half the

entire resources could have further room for improvement and use the additional window size to extract more instruction-level-parallelism and memory-level-parallelism.

All other GAP benchmarks have reached the instruction window saturation point and thus do not see any benefit from bigger windows. We also notice a slight dip in performance for the astar benchmark beyond a point due to the increased number of load violations in bigger instruction windows. The SPEC2017 benchmarks, as seen earlier, almost never succeed in constructing a helper thread and thus they do not see any effect with increased window sizes.

6.2.6 Sensitivity to input datasets

We tested out the sensitivity of Phelps to a different input dataset. We ran Phelps on the roadNet-TX dataset from [LK14]. Figure 6.17 shows the speedup normalized over baseline for Phelps on the roadNet-CA and roadNet-TX datasets.

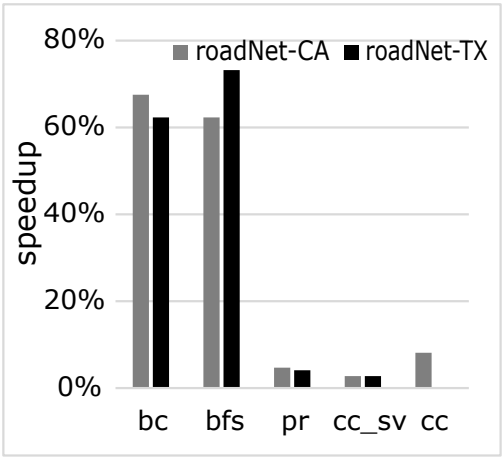


Figure 6.17: Performance of Phelps on different datasets

From the figure, it can be observed that Phelps shows similar trends of huge performance gains in benchmarks like *bc* and *bfs* (62% and 73% respectively). *Pr* and *cc_sv* replicate similar trend of a slight speedup just as in roadNet-CA. However, it is indeed true that

Phelps could be sensitive to the choice of input datasets. This can also be seen in the fact that the helper thread construction fails to construct a helper thread in the cc benchmark. This is due to the helper thread not being much shorter than the originally delinquent loop.

6.2.7 Sensitivity to pipeline depth

We also experiment with the depth of the processor's pipeline. Figure 6.18 shows the speedup of Phelps over similarly deep baselines on all benchmarks for depths 11, 15, and 19.

As the pipeline depth increases, speedups increase for benchmarks *bc*, *bfs*, *cc*, and *astar*. This is because, while both Phelps and the baseline have to endure the effects of a deeper pipeline, the baseline is affected more due to frequent branch mispredictions. Phelps, on the other hand, has streamlined execution: the helper thread is streamlined due to the absence of control-flow instructions and the main thread is more streamlined due to having accurate predictions. However, in benchmarks like *pr* and *cc_sv*, where Phelps barely manages to attain a slight speedup with the 11-deep configurations, increasing the pipeline depth further causes degradation in performance due to less timely predictions. The SPEC2017 benchmarks *mcf* and *x264*, which are the only ones that end up constructing a successful helper thread, show a minute increase in performance with deeper pipelines.

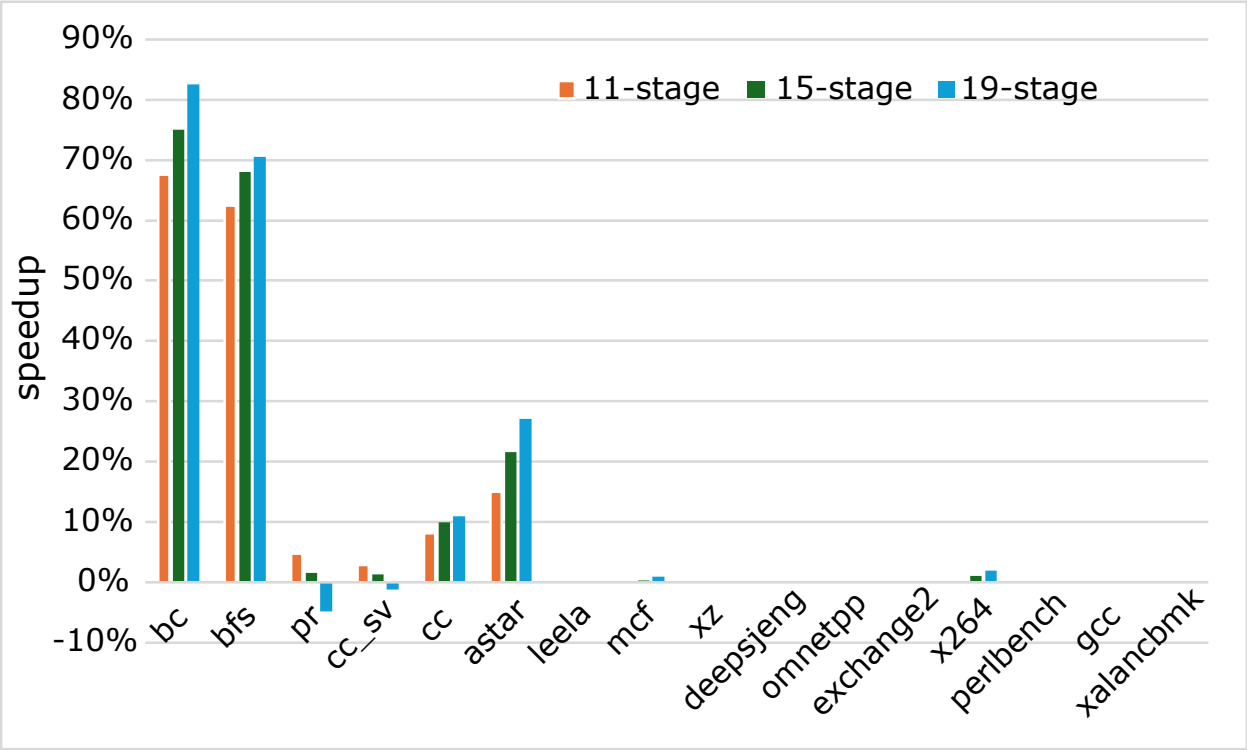


Figure 6.18: Sensitivity of Phelps to processor pipeline depth.

Chapter 7

Summary

Conventional history-based branch predictors are inadequate in accurately predicting the direction of data-dependent conditional branches. Branch pre-execution is a well-explored technique to handle delinquent branches. This is achieved by pre-executing the computation leading up to the delinquent branch ahead of time using redundant helper thread(s). However, prior works in branch pre-execution are ineffective in handling cases where a delinquent branch is control-dependent on another delinquent branch and/or when the backward slice of the targeted delinquent branch contains a store. This document presents my research that specifically focuses on addressing the above issues by unconditionally pre-executing nested delinquent branches and predicating the stores if they are guarded by delinquent branches.

My first contribution in this space is to deploy custom branch predictors in the Post-Fabrication Microarchitecture paradigm [KSC+21]. I showcase my custom branch predictor designs for the *astar* benchmark from SPEC 2006 and the *bfs* benchmark from the GAP benchmark suite. Both custom predictors essentially pre-execute delinquent branches in the region of interest, by mimicking the computation leading to these branches through a customized datapath. All delinquent branches are unconditionally pre-executed, including the control-dependent ones. Leveraging insights from the region of interest, the custom predictor correctly filters out excess predictions of control-dependent branches and infers the effects

of stores. At higher custom branch predictor bandwidth, the custom branch predictors are capable of producing huge speedups of up to 150% and 175% for the *astar* and *bfs* use-cases, respectively. The highly streamlined execution also results in a significant reduction in energy consumption.

My second contribution is Predicated Helper Threads (Phelps). Phelps is a fully-automated hardware-only branch pre-execution paradigm that focuses on pre-executing loops containing delinquent branches. Phelps dynamically detects delinquent branches and the loops that contain them, and constructs a helper thread for each such loop. All delinquent branches in the loop, including control-dependent ones, are unconditionally pre-executed. Per-branch prediction queues are managed in lock-step based on loop iterations. This enables (1) the helper thread to deposit outcomes of all delinquent branches for each loop iteration, and (2) the main thread to use or not use outcomes of a control-dependent branch based on the outcome of its guarding branch. Overall, this approach ensures that the pre-execution of nested delinquent branches is both timely and accurate. A helper thread also retains influential stores for dynamic disambiguation and store-load forwarding (without affecting architectural state of the main thread), and any such store that is control-dependent on a delinquent branch is predicated on that branch's outcome. A novel control-dependency (CD) analysis technique, based on simple hardware FSMs (that render reconvergent point detection unnecessary), is used to identify the branch that guards the store. Phelps also has the ability to create dual decoupled helper threads for outer-inner loop pairs to increase memory level parallelism. Phelps is observed to generate highly accurate and timely branch predictions, resulting in a geometric mean speedup of 13.8% across GAP benchmarks and *astar*. The speedup can be attributed to a mean 65% reduction in MPKI in these benchmarks.

REFERENCES

[AKPW83] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '83, page 177–189, New York, NY, USA, 1983. Association for Computing Machinery.

[ARS03] H. Akkary, R. Rajwar, and S.T. Srinivasan. Checkpoint processing and recovery: towards scalable large instruction window processors. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, pages 423–434, 2003.

[BAP15] Scott Beamer, Krste Asanović, and David Patterson. The GAP benchmark suite, 2015.

[CGS+20] Adarsh Chauhan, Jayesh Gaur, Zeev Sperber, Franck Sala, Lihu Rappoport, Adi Yoaz, and Sreenivas Subramoney. Auto-predication of critical branches. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 92–104, 2020.

[CHA+15] Trevor E. Carlson, Wim Heirman, Osman Allam, Stefanos Kaxiras, and Lieven Eeckhout. The load slice core microarchitecture. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 272–284, New York, NY, USA, 2015. Association for Computing Machinery.

[CSK+99] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt. Simultaneous subordinate microthreading (ssmt). In *Proceedings of the 26th International Symposium on Computer Architecture*, pages 186–195, May 1999.

[CTYP02] R. S. Chappell, F. Tseng, A. Yoaz, and Y. N. Patt. Difficult-path branch prediction using subordinate microthreads. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 307–317, May 2002.

[Fru20] Andrei Frumusanu. Apple announces the apple silicon m1: Ditching x86 - what to expect, based on a14. https://www.anandtech.com/show/16226/apple-silicon-m1-a14-deep-dive/2, November 2020.

[GH08] A. Garg and M. C. Huang. A performance-correctness explicitly-decoupled architecture. In *Proceedings of the 41st International Symposium on Microarchitecture*, pages 306–317, November 2008.

[HP15] Milad Hashemi and Yale N. Patt. Filtered runahead execution with a runahead buffer. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 358–369, 2015.

[KAGC98] A. Klauser, T. Austin, D. Grunwald, and B. Calder. Dynamic hammock predication for non-predicated instruction set architectures. In *Proceedings. 1998*

International Conference on Parallel Architectures and Compilation Techniques (Cat. No.98EX192), pages 278–285, 1998.

[KH19] Sushant Kondguli and Michael Huang. R3-dla (reduce, reuse, recycle): A more efficient approach to decoupled look-ahead architectures. In *Proceedings of the 25th International Symposium on High-Performance Computer Architecture*, pages 533–544, February 2019.

[KR04] J.J. Koppanalil and E. Rotenberg. A simple mechanism for detecting ineffectual instructions in slipstream processors. *IEEE Transactions on Computers*, 53(4):399–413, 2004.

[KSC+21] Chanchal Kumar, Anirudh Seshadri, Aayush Chaudhary, Shubham Bhawalkar, Rohit Singh, and Eric Rotenberg. Post-fabrication microarchitecture. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 1270–1281, New York, NY, USA, 2021. Association for Computing Machinery.

[LK14] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[MHM+95] Scott A. Mahlke, Richard E. Hank, James E. McCormick, David I. August, and Wen-Mei W. Hwu. A comparison of full and partial predicated execution support for ilp processors. *SIGARCH Comput. Archit. News*, 23(2):138–150, may 1995.

[MPV93] Mayan Moudgill, Keshav Pingali, and Stamatis Vassiliadis. Register renaming and dynamic speculation: an alternative approach. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, MICRO 26, page 202–213, Washington, DC, USA, 1993. IEEE Computer Society Press.

[PH14] R. Parihar and M. C. Huang. Accelerating decoupled look-ahead via weak dependence removal: A metaheuristic approach. In *Proceedings of the 20th International Symposium on High-Performance Computer Architecture*, pages 662–677, February 2014.

[PP21] Stephen Pruett and Yale Patt. Branch runahead: An alternative to branch prediction for impossible to predict branches. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 804–815, New York, NY, USA, 2021. Association for Computing Machinery.

[PSR00] Zach Purser, Karthik Sundaramoorthy, and Eric Rotenberg. A study of slipstream processors. In *Proceedings of the 33rd International Symposium on Microarchitecture*, pages 269–280, December 2000.

[PSR02] Zach Purser, Karthik Sundaramoorthy, and Eric Rotenberg. Slipstream memory hierarchies. Technical report, North Carolina State University, 2002.

[Rot08] A. Roth. Physical register reference counting . *IEEE Computer Architecture Letters*, 7(01):1–1, January 2008.

[RS01] Amir Roth and Gurindar S. Sohi. Speculative data-driven multithreading. In *Proceedings of the 7th Annual IEEE International Symposium on High-Performance Computer Architecture*, HPCA '01, pages 37–48, 2001.

[SCR20] Vinesh Srinivasan, Rangeen Basu Roy Chowdhury, and Eric Rotenberg. Slipstream processors revisited: Exploiting branch sets. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 105–117, 2020.

[Sez16] André Seznec. Tage-sc-l branch predictors again. In *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, June 2016.

[SKB+15] Manjunath Shevgoor, Sahil Koladiya, Rajeev Balasubramonian, Chris Wilkerson, Seth H. Pugsley, and Zeshan Chishti. Efficiently prefetching complex address patterns. In *Proceedings of the 48th Annual International Symposium on Microarchitecture*, MICRO-48, page 141–152, New York, NY, USA, 2015. Association for Computing Machinery.

[SPHC02] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. *SIGOPS Oper. Syst. Rev.*, 36(5):45–57, oct 2002.

[SPR00] Karthik Sundaramoorthy, Zach Purser, and Eric Rotenberg. Slipstream processors: Improving both performance and fault tolerance. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257–268, November 2000.

[ZS01] Craig Zilles and Gurindar Sohi. Execution-based prediction using speculative slices. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ISCA '01, page 2–13, New York, NY, USA, 2001. Association for Computing Machinery.