Maintaining High Performance in the Presence of Impossible-to-Predict Branches

Stephen Pruett

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

TR-HPS-2022-001
May, 2022

The Dissertation Committee for Stephen Pruett
certifies that this is the approved version of the following dissertation:

Maintaining High Performance in the Presence of Impossible-to-Predict Branches

Committee:

Yale Patt, Supervisor

Mattan Erez

Mohit Tiwari

Christopher J. Rossbach

Robert S. Chappell

Maintaining High Performance in the Presence of Impossible-to-Predict Branches

by

Stephen Pruett

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2022

Acknowledgments

Completing this dissertation would not have been possible without the love and support of my family, friends, and fellow graduate students. There are so many people that helped me in so many ways throughout this journey and there is simply not enough space to thank everyone. However, I would like to thank a few people here.

First, I thank my mother, Margaret Pruett, without whom I would not be where I am today. My mom always gave me the help and support I needed to succeed academically. She required that I participate in extracurricular activities and supported me as my ever changing interests caused me to bounce from sports to music to film. She enrolled me in programs that helped me build a strong academic foundation, which ultimately helped me become a successful engineer. She also supported me financially, allowing me to choose my path without the burden of financial decisions. Mom, it is not possible to thank you enough for all you have done for me over the years. I know you were hoping that I would one day become a music major. I hope that my Ph.D. in computer engineering will suffice.

I thank my father and step mother, Jeff and Gayla Pruett, for the support they have given me. They have always visited me while I was away on internships and called me to check on how I was doing. In particular, they

are always quick to come up with a plan for getting some much needed rest and relaxation. My dad has taught me so many things over the years, but in particular I think back to all the projects we have worked on together. When I was a kid, the first thing I ever said I wanted to be was a builder. My dad helped me cultivate that talent and creativity, which no doubt played a role in me eventually becoming an engineer. Gayla has always been incredibly supportive and patient, as at least a few of these projects have remained unfinished in her backyard. They have also supported me financially, which helped me pursue my dreams.

I thank my sister and brother-in-law, Lauren and Cyrus Manekshaw. My sister was a roll model while growing up and has always been quick to drop whatever is going in her life to help me when I need her most. I consider myself very lucky that Cyrus found his way into our family. His kindness and sense of humor fit right in, and I of course always appreciate our late night runs to Andy's. I look forward to watching their children, Zubin and Zara, grow up and perhaps one day reading their dissertations.

I thank my grandparents, Tom and Jo Phalen and Jerry and Pat Pruett. I still remember working on my multiplication tables with my Grandpa Phalen, reading books with my Grandma Phalen, playing card games with my Grandma Pruett (somehow I always won!), and working on projects in my Grandpa Pruett's garage. Given that I am the grandchild of two teachers and two engineers, it is no wonder where my passion for both engineering and teaching has come from.

I thank past and present members of HPS, including Aater Suleman, Carlos Villavieja, Jose Joao, Rustam Miftakhutdinov, Milad Hashemi, Faruk Guvenilir, Ben Lin, Siavash Zangeneh, Ali Fakhrzadehgan, Aniket Deshmukh, and Chester Cai. Each of them are always willing to stop what they are working on and help with whatever challenge I may be facing, whether it is related to research or not. They provide an environment of openness, candidness, and support that makes for an ideal research environment. In particular, I would like to thank:

- Aater, for the help and graduate school advise you gave me during my first year as a graduate student. Your ability to brainstorm and break down complex research ideas was an early inspiration for me. Your perspective on how to approach graduate school was something I reflected back on often, and recently your perspective on joining a start up helped me decide what I will do next.

- Carlos, for working with me throughout my senior year of undergrad and my first year in graduate school. You helped be get started with research and to eventually become a member of HPS.

- Jose, for your advise and kindness, both early on in my Ph.D. and recently while looking for a job.

- Rustam, for looking over my shoulder during my first year and asking me tough questions about what I was working on. Although you may not

realize it, your inclusiveness during my first year helped me acclimate to graduate school and working with HPS. Your attention to detail and tough questions have improved my research over the years. You have also helped in many situations where you did not need to, including proof reading papers, teaching me how to use Scarab, and dry running my first presentation to Yale. I also thank you for your mentorship during internships and teaching me how to play foosball.

- Milad, for his brutally honest feedback and willingness to tell me when my research was headed in the wrong direction. Although I did not always immediately listen, your advise was often correct and eventually led me in the right direction. I also appreciated your willingness to share a bottle of North Korean wine with me.

- Faruk, for patiently and painstakingly teaching me interrupts, virtual memory, and a variety of other topics throughout my undergrad as well as many other things over the years. You are one of the best teachers I know, and it is something that has truly been an inspiration to me. I always appreciated discussing research with you, as your natural teaching instincts and desire to deeply understand everything always left me understanding my own research that much better.

- Ben, for the conversations on our walks home after a long night of research. Your openness and friendship helped me fully acclimate and find my place in the research group. You have always been around and

willing to help out with my research, and your help proof reading my papers has been essential. Your good attitude and sense of humor always brightened the office and made our group fun to work in.

- Siavash, for being my partner in all things branch prediction. Your deep understanding on a variety of topics as well as your creativity unlocked a number of interesting research projects that I am extremely proud to be a part of. You have taught me so many things and helped me in so many ways over the years that it is hard to list them out here. I am particularly grateful for our time TAing together, the late nights working on the branch prediction competition together, and all of the interesting topics that I got to research with you. Also for your deep appreciation of Dr. Pepper and Chick-fil-A.

- Ali, for our walks, good discussing, and of course always joining me in the longhorn run. I particularly appreciate the time you spent helping me with my papers and teaching me Farsi.

- Aniket and Chester, for taking up the mantle in HPS. Getting the opportunity to see each of you develop and show off your creativity has been inspiring.

I thank Austin Harris, Taylor Morrow, David Knopf, and Ali Mohandesi for being the best study group anyone could ask for during our undergrad. Our countless late nights and hard work helped me build the strong foundation in

computer engineering that I relied on all through out graduate school. Having y'all as partners enabled me to take on so many challenges in undergrad that I am still extremely proud of today. Additionally, I thank Austin for being a fantastic roommate and for having to put up with me slightly more than everyone else, Taylor for your encyclopedic knowledge on all topics as well as your continued friendship, David for bringing us all together, and Ali for making sure we always took time to have fun.

I thank my fellow graduate students, including Jeremie Kim, Zach Myers, and Swamit Tannu for your friendship and guidance throughout graduate school.

I thank my friends Matt Brown, Sam Chacon, Laura Condon, Stafford Hutchins, Adrian Orozco, Corey Rose, Manuel Santiago, Spencer Walker, and Blake Wolf. I appreciate the efforts you all go through to make sure we have continued to stay in touch. Our long and continued friendship is something that I deeply appreciate.

I thank my internship managers, Rob Chappell, Belli Kuttanna, Burton Smith, Doug Carmean, Tom Huff, and Kulin Kothari for giving me great learning opportunities in industry. In particular I would like to thank Rob Chappell for the many conversations and good advice he has given me over the years.

I also thank the ECE administrative staff, in particular Leticia Lira for her outstanding support of our research group, Melanie Gulick for her quick

and attentive responses and always making sure I had all the right forms, and Melody Singleton for her hard work and help during ECE visit days.

I thank Mattan Erez, Mohit Tiwari, Chris Rossbach, and Rob Chappell for serving on my dissertation committee and for the valuable feedback that improved this dissertation.

Finally, I thank my advisor, Yale Patt. Like many of his students, my first exposure to Yale was in his EE 306 class, Introduction to Computing. It was this course that started my interest in computer architecture. Additionally, it was Yale's approach to teaching that inspired me to build a strong foundation and never put a ceiling on what I could accomplish. He challenged me to strive for a well rounded education, not just focusing on math and engineering, but also focusing on the importance of good writing and communication skills. Yale sets the bar high, and expects you to measure up. An exercise that can be challenging, but one that I am very grateful for. I am most inspired by his amazing ability to teach and get the most out of his students. That is a quality that I hope I can one day achieve myself.

Stephen Pruett

May 2022, Austin, TX

Maintaining High Performance in the Presence of Impossible-to-Predict Branches

by

Stephen Pruett, Ph.D.
The University of Texas at Austin, 2022

SUPERVISOR: Yale Patt

High performance microprocessors have relied on accurate branch predictors to maintain high instruction supply for over 30 years. Unfortunately, as instruction windows and pipeline widths have continued to grow, misprediction penalties have gotten worse. Branch predictors have failed to improve at a fast enough rate to counteract these penalties. Impossible-to-predict branches, such as data-dependent branches, have become the worst offender since, so far, no viable predictor exists for these branches. I propose to identify such branches at runtime, and replace the inaccurate branch prediction with a more accurate merge point prediction. Doing so enables techniques that can either pre-compute the result of the branch, as is the case for Branch Runahead, or avoid the misprediction altogether by dynamically predicating instructions, or fetching instructions out-of-order; i.e., from the merge point until the branch direction has been determined. This dissertation presents a

new merge point prediction algorithm that achieves a higher accuracy and coverage than prior work, and uses it to enable three mechanisms for dealing with impossible-to-predict branches: Branch Runahead, Dynamic Predication, and Delayed Fetch.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 The Problem: The Conditional Branch Bottleneck

High performance microprocessors require high levels of instruction supply. As instruction windows and fetch rates continue to grow, the demand for higher levels of instruction supply rises. Continued improvements in branch prediction accuracy have been the most important driver of this for nearly 30 years. Unfortunately, modern predictors are increasingly bottlenecked by hard-to-predict branches. While branch prediction accuracy will continue to improve in the future, the rate of improvement is being outpaced by the demand for increased instruction supply. Unfortunately, there are many branches, such as data-dependent branches, that are impossible for traditional history-based predictors to predict accurately. In these cases, branch prediction will always fall short. Despite this, branch prediction remains the only runtime solution for conditional branches.

Existing microarchitectures must predict all conditional branches, even ones on which they have performed poorly in the past. This leads to high branch misprediction rates in the presence of branches that are fundamentally hard-to-predict. Each misprediction corresponds to an expensive pipeline flush

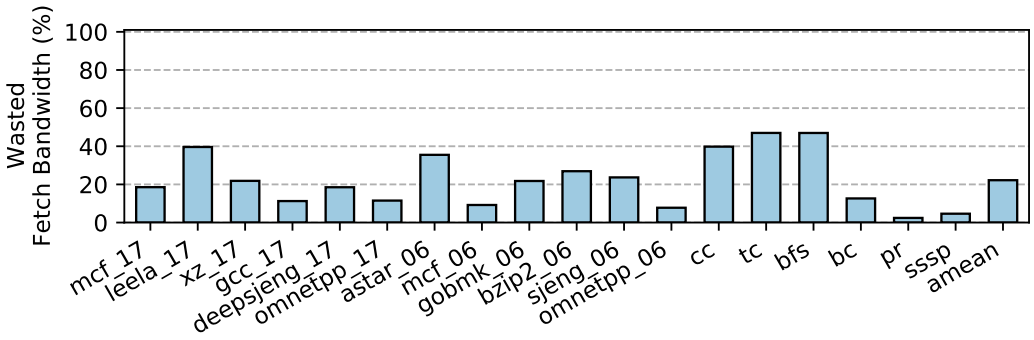

Figure 1.1: Percentage of processor cycles wasted due to fetching wrong-path instructions. Table 4.1 contains the system configuration.

that lowers the effective fetch rate and wastes energy. Figure 1.1 shows the percentage of processor cycles wasted due to fetching wrong-path instructions as a result of branch mispredictions. Each of these cycles, at a minimum, corresponds to wasted power and energy, and often also results in significant performance loss. This all-in approach on branch prediction is particularly problematic when faced with branches that are impossible-to-predict. In these cases, the processor has no ability to dynamically move from branch prediction to alternative methods that may provide higher performance or greater energy efficiency.

Alternatives to branch prediction, such as pre-computation and control-independence, have been proposed; however, these techniques have limitations, such as heavy dependence on a compiler or high hardware costs that discourage implementation. This dissertation presents a holistic new approach to branch prediction alternatives. To do this, I improve upon essential structures

required by branch prediction alternatives, such as hard-to-predict branch detection and merge point prediction. Once the hard-to-predict branches and their merge points have been identified, I use either pre-computation (Branch Runahead) or control independence (Dynamic Predication and Delayed Fetch) to avoid branch prediction altogether.

1.2 Impossible-to-Predict Branches

Impossible-to-predict branches are branches for which there is currently no viable predictor that can predict the branch accurately. I.e., the branches are *impossible* on a deep algorithmic level for a given branch predictor, beyond just predicting the bias of the branch. An example of this is data-dependent branches for the TAGE-SC-L predictor [52]. The TAGE predictor offers no ability to specifically target this class of branches beyond predicting the bias of the branch. Therefore, these branches often perform poorly on the TAGE branch predictor. In such cases, simply making the predictor larger will not improve accuracy because the predictor itself is not capable of learning useful information about the branches. The fact that the branch predictor has no ability to learn to predict these branches accurately motivates the branch prediction alternatives presented in this dissertation.

Detecting branches that are truly impossible-to-predict, however, is impractical. Instead, this dissertation detects *hard-to-predict branches*; i.e., branches that TAGE-SC-L performs poorly on. This is done by adding a new structure, the Hard Branch Table (HBT). A branch is contained in the HBT if

its misprediction rate is higher than a given threshold. If a branch hits in the HBT, this means that the branch predictor (for whatever reason) is not able to predict the branch accurately and a branch prediction alternative, such as Branch Runahead, Dynamic Predication, or Delayed Fetch, should be used in its place. If a branch misses in the HBT, it is assumed that the branch is being predicted accurately and the branch will continue to use the branch predictor.

1.3 Detecting Hard-to-Predict Branches

The branch prediction alternatives proposed in this dissertation should only be used in cases where the traditional branch predictor fails to achieve high accuracy. Therefore, the first step this dissertation takes is to identify branches for which the branch predictor is performing poorly. The first method discussed detects branches whose misprediction rate exceeds a target threshold. However, as will be shown in this dissertation, not all branch mispredictions are created equal. Long latency branch mispredictions can have a higher impact on performance. In extreme cases, branches that mispredict relatively infrequently can still become bottlenecks if their result depends on a long latency operation, such as a cache miss. To handle these cases, we augment our methodology to also account for branch latency.

Prior work in branch confidence estimation can be categorized as having at least one of the two following issues. First, many prior techniques are *path-based* confidence techniques [51, 19, 4, 36, 20]. The goal of these techniques is to identify the path, or the specific dynamic instance of a branch that is

likely to mispredict. These techniques are really intended to identify specific branch predictions that have low-confidence and throttle only those instances. The techniques presented in this dissertation, however, work on a per-branch basis. These techniques require expensive transformations that are only worth it for branches that contribute many mispredictions throughout the entire program. Second, many prior techniques are too dependent on recent branch behavior [26, 51, 55, 19]. For example, the JRS confidence predictor [26] resets its confidence counter on each misprediction, classifying a branch as low confidence if it has mispredicted recently. The techniques presented in this dissertation benefit more from a more stable view of the hard-to-predict branches. Ideally, we would like to generate a list of the top N hard-to-predict branches in the workload and only update that list if another branch proves to be in that top N.

To address these shortcomings, I introduce the Hard Branch Table (HBT). The HBT is a small table that monitors the retired instructions stream for branch mispredictions. The table uses a Leaky Bucket counter [24] to detect branches whose misprediction rate exceeds a given threshold. An arithmetic model is used to compute the parameters of the HBT based on the target misprediction rate and an acceptable false positive rate. Furthermore, Chapter 2 discusses how to extend the HBT to account for branch waste; i.e., the impact each branch misprediction has on performance. With this, the HBT is used to detect which branches will benefit the most from Branch Runahead, Dynamic Predication, and Delayed Fetch.

1.4 Merge Point Prediction

An important contribution of this dissertation which is necessary for the alternatives to branch prediction is a much more accurate merge point predictor than anything available prior to this dissertation. Prior approaches use compiler heuristics and assumptions about code layout to predict the location of the merge point that do not reliably locate the best merge point. My work takes advantage of branch mispredictions by comparing instructions fetched from both the wrong path and the correct path to detect the merge point. This approach is more accurate and reliable than prior work.

The merge point predictor is able to achieve an average accuracy of 95% across the SPEC CPU2006 benchmark suite [3]. The improved accuracy results in successfully detecting and replacing 58% of all branch mispredictions with a correct merge point prediction, reducing the MPKI by an average of 43%. This dramatic improvement in coverage gives us more opportunity to substitute an alternative for branch prediction in cases where the branch predictor performs poorly.

1.5 Pre-computation: Branch Runahead

Branch Runahead is a mechanism that uses lightweight dependence chains to pre-compute the result of hard-to-predict, data-dependent branches. Prior work used the compiler to create a filtered version of the original program, only containing instructions necessary to compute the result of hard-to-predict branches. The filtered thread, or "helper" thread, is executed asyn-

chronously on another core [59, 31], on another Simultaneous Multi-threading (SMT) context [64, 49, 11, 10], or on a dedicated unit within the core [54]. These approaches are fundamentally more costly as they require use of the compiler and/or re-executing most instructions (85%) in the program, and thereby require expensive resources to pre-compute the branch.

Dependence chains are far simpler than the helper threads proposed by prior work, which allows them to be accelerated on a small, simple hardware structure, rather than another core or SMT context. Branch Runahead focuses on predicting hard-to-predict branches; E.g., branches on which TAGE-SC-L [52] performs poorly. In particular, Branch Runahead is ideal for data-dependent branches that have a small number of instructions required to compute the predicate. Branch Runahead, when configured under reasonable hardware constraints, reduces branch MPKI by 47.5% and increases IPC by an average of 16.9%.

1.6 Control Independence

1.6.1 Dynamic Predication

Dynamic predication is a runtime mechanism for fetching both paths of a branch, up to the merge point, then executing each path and only committing the results of the correct path. Predication avoids the costly misprediction penalty while also guaranteeing that the correct path is always fetched, decoded, renamed, and executed, even if the branch in question has not completed execution. Once the result of the branch is known, the correct

path results are passed on to any dependent, waiting, post-merge point instructions. In cases where branch mispredictions are frequent, this model can minimize the length of the critical path, as both paths are always executed.

Unfortunately, the wrong path is also always fetched, decoded, renamed, and executed. In some cases, the wasted fetch and execute bandwidth from the wrong path is too great, causing performance inversions; i.e., cases where the performance of branch prediction exceeds that of dynamic predication. Furthermore, predication adds new data dependencies to the data-flow graph, which can increase the length of the critical path and worsen performance.

Prior Dynamic Predication techniques require expensive hardware to track the micro-architectural state of correct-path and wrong-path instructions separately [28]. Other approaches sacrifice early execution of predicated instruction to simplify these hardware demands [12]. My approach to Dynamic Predication will focus on low-cost hardware while still allowing predicated instructions to execute early.

1.6.2 Delayed Fetch

Delayed Fetch is a runtime mechanism that does not fetch instructions from either path of the branch. Instead, Delayed Fetch fetches instructions out-of-order from beyond the merge point. Once the branch has executed and the correct path of the branch is known, Delayed Fetch fetches the correct path of the branch and inserts the delayed instructions into the instruction

window in their correct program order.

The primary benefit of Delayed Fetch is conserving instruction fetch bandwidth since only correct path instructions are fetched. This, however, comes at the expense of correct path latency since correct path instructions are not fetched until the result of the branch is known. This increase in correct path latency can also lead to performance inversions.

Prior approaches only address Delayed Fetch for simple if-then branches [13]. Furthermore, prior work does not address predicting branches out of order, which limits the technique to skipped regions that either do not contain any branches or only contain biased branches. My approach to Delayed Fetch introduces a new technique for handling the branch predictions in the skipped region. This not only enables Delayed Fetch to be used in more cases, but also simplifies other issues like out-of-order Rename.

1.6.3 Combining Dynamic Predication and Delayed Fetch

Ultimately, Dynamic Predication reduces the latency of correct path instruction at the expense of fetch, decode, and execute bandwidth, while Delayed Fetch maximizes fetch, decode, and execute bandwidth at the cost of correct path instruction latency. While neither Dynamic Predication nor Delayed Fetch are perfect solutions on their own, the reciprocal nature of their trade-offs make them perfect complements of one another. This dissertation explores combining Dynamic Predication and Delayed Fetch. As such, we propose a single microarchitecture that efficiently implements both techniques.

Further, we demonstrate switching between the techniques on a per application basis. This allows the technique that is most suited towards the needs of the application to be selected. Finally, techniques to switch on a per-branch basis are discussed in the future work section of the chapter.

1.7 Thesis Statement

Dynamic merge point prediction enables effective branch prediction alternatives for impossible-to-predict branches allowing the architecture to dynamically choose between pre-computation or control-independence, thereby avoiding branch mispredictions, resulting in increased performance and energy efficiency.

1.8 Contributions

The contributions of this dissertation are as follows:

- A new method for detecting hard-to-predict branches. This dissertation presents an arithmetic model that can be used to detect branches that mispredict above a given frequency.
- Further, this model is extended to detect branches that have high cost (i.e., flush a large number of instructions), which is more directly related to performance than misprediction rate alone.
- A new merge point prediction algorithm, which improves both accuracy and coverage over prior work.
- Branch Runahead, a new light-weight pre-computation technique that

generates near perfect branch predictions for data-dependent branches. Branch Runahead improves on prior work by lowering the dynamic instruction footprint of the pre-computation, resulting in a lower hardware and energy overhead.

- A holistic new approach to Dynamic Predication and Delayed Fetch. This dissertation presents a unified micro-architecture that solves critical issues related to both Dynamic Predication and Delayed Fetch. Further, we present an analysis of both positive and negative results to discuss what areas of control independence need further improvement.

1.9 Dissertation Organization

This dissertation is organized into six chapters. Chapter 2 discusses hard-to-predict branch detection. The chapter also defines branch cost, explains why it is more closely related to performance than mispredictions alone, and expands hard branch detection to include branch cost. Chapter 3 discusses the merge point prediction. This includes merge point detection, prediction, and predictor update. Chapter 4 discusses Branch Runahead, a light-weight pre-computation technique for Branch Prediction. The chapter also shows how the merge point predictor can be extended for affector and guard branch detection- a technique that Branch Runahead uses to identify control and data dependencies between branch instructions. Chapter 5 discusses Dynamic Predication and Delayed Fetch, as well as a new holistic micro-architecture that solves critical issues with both. Chapter 6 provides concluding remarks.

Chapter 2

Hard-to-Predict Branch Detection

2.1 Introduction

This dissertation presents three branch prediction alternatives, Branch Runahead, Dynamic Predication, and Delayed Fetch, that are used when a branch is deemed too hard for the on chip branch predictor to predict. The hard-to-predict branches go through an expensive chain extraction (in the case of Branch Runahead) or code transformation (in the case of Dynamic Predication and Delayed Fetch) process in order to use these techniques. In the case of Branch Runahead, only a limited number of branches (16 branches) can use the pre-computation resources. For these reasons it is important to be selective and decisive when choosing branches for these optimizations. Prior work in branch confidence estimation does not produce stable enough results. Furthermore, prior work is not able to account for the total cost of a branch misprediction. This chapter introduces the Hard Branch Table (HBT)[1], a new structure that detects branches that mispredict at or above a given threshold. In this chapter, the design of the Hard Branch Table is discussed, as well as the arithmetic model that is used to configure the table. Finally, the chapter

[1] A version of the HBT was first introduced in the Branch Runahead [45] paper, which is my own work.

is concluded by discussing Branch Cost, a new metric that tracks the true impact of each branch misprediction on overall performance, and the extensions required to track Branch Cost in the HBT.

2.2 Limitations of Prior Work

Jacobsen et al. introduced several confidence mechanisms [26]. Most notably was their confidence mechanism using a simple PC-indexed table of 3-bit counters. The counters are incremented on a correct prediction, and reset on a misprediction. Due to the aggressiveness of resetting the counter on a misprediction, a saturated counter implies a very high degree of confidence. Unfortunately, the counter values can also be unstable, because one misprediction can reset a long history of correct predictions. Further, the counters are very susceptible to phase behavior, meaning that if a branch changes phase, the counters will be reset and confidence must once again be earned. The Hard Branch Table, on the other hand, assumes that some mispredictions occur. In fact, the table is designed in such a way that it can tolerate mispredictions as long as they remain below the target rate. This keeps the classification of hard branches very stable throughout the program. If a branch truly changes phase, turning into an easy branch, it must demonstrate that behavior across multiple periods. This behavior is desirable, as branch prediction alternatives often require a warm up cost before activating. Therefore, stable behavior is preferred.

Seznec proposed a storage-free method for estimating confidence of pre-

dictions from the TAGE branch predictor [51]. While this method is undoubtedly effective and low overhead, it is tied specifically to the TAGE branch predictor. Moreover, this method associates a level of confidence with a particular prediction, rather than generating a cumulative per branch confidence. Currently, all branch prediction alternatives are either activated or deactivated on a per-branch basis. Meaning that, in the context of this dissertation, getting a per-prediction confidence is not useful. In future work, I believe it would be possible to combine the Hard Branch Table together with the Seznec's storage free method to more selectively apply Dynamic Predication and Delayed Fetch techniques (discussed later in this dissertation).

2.3 Detecting Branches with High Misprediction Rates

The simplest way to find hard-to-predict branches is by looking at the misprediction rate of retiring branch instructions. We look at misprediction rate (i.e., mispredictions per kilo-instruction) rather than misprediction ratio (i.e., the percentage of the time that a given branch mispredicts) because ultimately we care about minimizing the number of pipeline flushes per kilo-instruction, and misprediction rate directly corresponds to the flush rate.

There are two ways to define hard-to-predict branches. The first is to say that the hard-to-predict branches of a program are the N branches with the highest misprediction rates. This methodology, however, would classify branches as hard-to-predict even if they did not cause very many pipeline flushes, simply because they happened to be in the top N hard to predict

Variable	Definition
Period (D)	Length of a period, measured in retired branch mispredictions.
Acceptable Misprediction Rate (AMR)	An arbitrary threshold where any branch mispredicting above threshold is considered hard-to-predict.
Probability of a False Positive (PFP)	An acceptable probability of a false positive; i.e., the probability that a counter saturates for a branch that is not hard-to-predict.
p	The probability of a misprediction.

Table 2.1: Definition of variables for the Arithmetic Model.

branches in the workload.

Instead we classify hard-to-predict branches based on whether they are above or below a given misprediction rate threshold. The idea being that for every pipeline of a given width and depth there is some level of acceptable flushes per kilo-instruction. As long as the MPKI stays at or below the rate of acceptable flushes, then we do not have a performance/energy problem.

Therefore the goal of the Hard Branch Table is to detect branches that rise above this acceptable rate. Any branch above this threshold may cause problems for performance.

2.3.1 The Arithmetic Model

To achieve this, I developed a simple arithmetic model to describe hard branch behavior. I then design a table, the Hard Branch Table, that detects hard branches based on the arithmetic model.

To start, lets consider each static branch individually. The goal is to

determine if a given static branch has a misprediction rate higher than the given acceptable misprediction rate (AMR). To achieve this, I divide time into periods. To keep the math simple, I choose to measure time in retired mispredicted branches.[2] In the general case, the length of the period is D, but an example value of D used throughout this dissertation is D=1000 retired mispredicted branches.

Dividing time into periods allows us to break up the problem into two pieces: 1) what happens across periods and 2) what happens within the period.

What happens across periods. The HBT uses Leaky Bucket counters [24] to detect if a branch mispredicts at a rate higher than the AMR. As mispredicted branches retire, a counter stored in the HBT is incremented. The counter is decremented periodically at a rate equal to the AMR. In this scheme, all branches that have a misprediction rate higher than the AMR will (eventually) saturate their counters (i.e., fill the Leaky Bucket), while branches that have a misprediction rate lower than the AMR will eventually saturate to 0 (Empty Bucket). The simplest way to decrement the counter at this rate is to decrement at the end of each period by $AMR \times D$.

$$DecrementRate = AMR \times D$$

What happens within the period. Decrementing the counter at

[2]Time can be measured by counting the number of retired instructions, retired branch instructions, or retired mispredictions. However, the AMR must be adjusted accordingly.

the end of each period only works as expected if the counter does not saturate during the given period. If the counter were to saturate during the period, this would throw off the effective increment rate, which could potentially increase the impact of the decrement rate, leading to poor classification of hard-to-predict branches.

Therefore, we must ensure that the counter is wide enough (i.e., enough bits) such that the probability of saturating the counter *in one period* is sufficiently low. This probability is refered to as the *probability of a false positive (PFP)*. To find the desired width we must first compute the probability of saturating the counter. To compute that, we model the sequence of mispredicting branches as a Binomial Distribution. The Binomial Distribution is used to represent the string of mispredicted/correctly predicted branches within the period (1 means mispredicted, 0 means correctly predicted). Each Binomial computes the probability that i mispredictions occurs. We then sum these probabilities for all i large enough to saturate a k-bit counter.

$$CounterBits = log_2(k+1)$$

for the smallest value k such that

$$PFP > \sum_{i=k}^{D} \binom{D}{i} p^i (1-p)^{(D-i)}$$

This arithmetic model leads to counters that are wide enough such that they have a very low likelihood of saturating spuriously, while also minimizing

the counter width to not create more hardware cost or warm-up time than what is necessary. The arithmetic model essentially transforms parameters like decrement rate and counter width, parameters that can feel arbitrary to assign values to, to parameters that are more human relatable, like acceptable misprediction rate (AMR) and probability of a false positive (PFP). This gives architects more control over the parameters of the HBT, without requiring simulation sweeps for fine tuning.

2.3.2 Hard Branch Table

The final resulting piece of hardware derived from the equations above turns out to be relatively simple. Throughout this dissertation, we use a Hard Branch Table that is 64 entries large. The table is indexed and tagged with the PC of the branch instruction. At each location, there is a 5-bit saturating misprediction counter. The misprediction counter is incremented each time a corresponding branch misprediction is retired. All counters are decremented by 15 after 1000 global branch mispredictions are retired. These numbers correspond to a Hard Branch Table with a AMR of 1.5%, a period of 1000, and a PFP of 1%.

2.4 Branch Cost

Branch mispredictions are not all created equal. To illustrate this, let's consider the following example.

Consider a theoretical machine that has a perfect i-cache, perfect d-

cache, perfect target predictor, 16-wide fetch/decode/rename, and contains 12 pipeline stages before execution. This same machine has a branch predictor that is 95% accurate. What is the maximum utilization of such a machine?

Assuming 1 out of every 5 instructions is a branch, and given that 1 out of ever 20 branches is a misprediction, we can therefore infer that 1 out of every 100 instructions will be a branch misprediction. With a 16-wide fetch unit we are able to reach the mispredicting branch after 7 cycles ($100/16 = 6.25$). Given that the pipeline contains 12 stages before execution, the minimum latency of the branch is 12 cycles. Therefore every 7 cycles there will be a 12 cycle period where we fetch no useful instruction. This puts machine utilization at $7/(7 + 12)$ or about 36.8%.

More generically, machine utilization can be given by the following ratio between useful cycle : wasted cycles.

$$Utilization = \frac{5(1-a)^{-1}}{W} : L$$

Where a is the branch predictor accuracy, W is the machine width, and L is the branch execution latency. This ratio makes clear why branch mispredictions increasingly become a bottleneck as machine width and depth increase. As machine width increases, we more quickly fetch our way through correct path instructions, reducing the total number of correct path cycles. While as the pipeline becomes deeper, we increase the minimum latency of a misprediction, thereby increasing the minimum latency of a branch misrpedic-

tion.

This example is also meant to illustrate that the waste created by a branch misprediction increases as the latency of the misprediction increases. Therefore, considering the accuracy of branch is only half the story. One must also consider a branch's expected latency to truly determine the expected branch misprediction cost.

2.5 Detecting Branches with High Branch Cost

Branch latency plays a key role in branch cost; however, latency and misprediction rate alone do not describe the full picture. Complex interactions in the pipeline, such as i/d-cache misses and full window stalls also impact branch cost. Deriving a formula that accounts for all involved variables is complex and unnecessary. Instead, branch cost can be measured directly by counting the number of micro-ops that are flushed from the Re-order Buffer (ROB) due to a retired branch misprediction.

Each flushed instruction represents a single unit of wasted fetch bandwidth due to the branch misprediction. The total number of flushed micro-ops represents the total wasted fetch bandwidth due to the branch misprediction and fully accounts for all complex pipeline interactions. The total number of flushed miro-ops can be quickly computed by subtracting the tail pointer for the ROB with the ROB ID of the branch instruction. Note that to truly account for all wasted fetch bandwidth, micro-ops flushed from fetch, decode, rename, and any other stages that come before ROB allocation should also be

accounted for. Counting this additional micro-ops, however, adds unnecessary complexity to the pipeline. Simply counting the micro-ops flushed from the ROB is sufficient for estimating wasted fetch bandwidth.

2.5.1 Branch Cost Table

Integrating branch cost into the HBT is a trivial task. Whereas before we treated all branch mispredictions equally, now each misprediction is weighted by the branch cost. When a branch misprediction occurs, the number of micro-ops flushed from the ROB is computed. Then, rather than incrementing the HBT misprediction counter by 1, the counter is incremented by branch cost. Furthermore, the decrement rate and the HBT counter width need to be increased to reflect the new increment rate.

2.6 Related Work

There are many other confidence estimations techniques not directly addressed in this dissertation. However, all techniques suffer from the issues described above. Jim Smith uses the magnitude of the saturating counter in the bimodal table to estimate branch confidence [55]. While this method is useful in that it requires no modification on top of a baseline branch predictor (and was later extended to TAGE [51]), this confidence mechanism is still limited to assigning confidence to particular predictions from the branch predictor. Further, because it is tied to the state of single bimodal table, it adapts too quickly to recent branch behavior. The accuracy of these technique can

be improved by using either a path history [19, 20, 36] or Perceptrons [4, 27]; however, these improvements do not address the limiting issues with these technique.

2.7 Future Work

One limitation of the Hard Branch Table is its small size. While this is not a problem for SPEC or Graph workloads, workloads that have larger instruction footprints may have issues fitting in the small HBT. Large instruction footprint can impact the HBT in two areas. First, it can impact the replacement policy of Hard Branches. Currently, branches are replaced when their misprediction counter is zero. However, in large instruction footprint applications the effective reuse distance of branches is much larger, which can make caching a set of them in a small cache more difficult. Therefore, a separate aging mechanism is needed to retain branches that have not been seen recently.

Second, a large instruction footprint can lead to a large number of branches that reach the minimum threshold requirements for making it in the HBT. Therefore, a dynamically adapting AMR threshold is needed to raise the threshold such that only the hardest to predict branches in the workload are able to enter into the table. In this model, the AMR would start off high. Occupancy of the table would then be monitored. If not enough branches are able to overcome the high threshold (and therefore the occupancy of the HBT is low), then the AMR will by dynamically lowered. This will allow more

branches into the table. This process will be repeated until the occupancy reaches a desired threshold.

Chapter 3

Merge Point Prediction

3.1 Introduction

This chapter discusses dynamic merge point prediction as a runtime alternative to branch prediction.[1] By predicting the merge point of a branch, the processor can avoid an expensive branch misprediction, instead utilizing a control independence strategy [13, 47, 25, 50, 56, 37, 14, 42, 43, 6, 48]. A control independence strategy is a technique that does not require knowledge of the branch direction, but can be used to mitigate or avoid a branch misprediction. This chapter proposes and evaluates a fundamentally new algorithm for detecting merge points. Prior approaches use compiler heuristics and assumptions about code layout to predict the location of the merge point that result in limited accuracy and coverage. My work takes advantage of branch mispredictions by comparing instructions fetched from the wrong path and correct path to detect the merge point. I argue this new approach to merge point prediction is fundamentally more accurate and reliable than prior work.

The merge point predictor is able to achieve an average accuracy of 95% across the SPEC CPU2006 benchmark suite [3]. The improved accuracy

[1] The ideas presented in this Chapter were first published in [44], which is my own work.

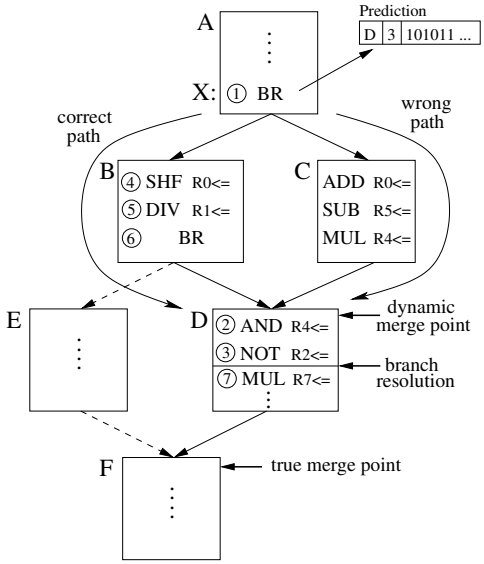

Figure 3.1: Example Control Flow Graph (CFG)

Dashed edges indicates paths that are rarely traversed at runtime. The compiler would report that the merge point of A is F. However, because block E is rarely seen, D is predicted as the merge point.

results in successfully detecting and replacing 58% of all branch mispredictions with a correct merge point prediction, reducing the MPKI by an average of 43%.

3.2 Motivation

The merge point predictor is designed with common control independence techniques in mind [13, 47, 25, 50, 56, 37, 14, 42, 43, 6, 48]. My work emphasizes three key principles that I believe to be essential for utilizing control independence effectively. First, only hard-to-predict or long latency branches are candidates for merge point prediction. We only consider merge

point prediction an option when a branch misprediction is too risky. Second, predicted merge points should be as close to the branch as possible. Often, the merge point predictor identifies more than one potential merge point for a given branch. This is because the merge point predictor is identifying *dynamic* merge points, which will be explained later in this section. Selecting merge points that are closer to the branch increases the number of post merge point instructions that are control and data independent of the branch. Furthermore, it decreases the number of resources that are required by the merge point, which reduces the size of reservations required by some control independence strategies. Third, merge point predictions must be accurate. If a merge point prediction is wrong, then the machine must be flushed, similar to a branch misprediction. We design a highly accurate dynamic merge point predictor that generates predictions at runtime without relying on compiler input or code layout[2]. Prior predictors [13, 15] make assumptions about compilers and code layout, making their work inaccurate and resistant to change. Our predictor takes advantage of branch mispredictions, finding the point where correct-path and wrong-path converge, making our predictor oblivious to compiler changes.

3.2.1 Weaknesses of Detecting Merge Points at Compile Time

The compiler itself could be used to easily identify merge points with 100% accuracy, however highly biased branches can weaken the compiler's

[2]Arrangement of basic blocks in memory

ability to find the nearest merge point [28, 15], which can negatively affect performance. Furthermore, identifying hard-to-predict branches at compile time is difficult, reducing the compiler's ability to provide help where it is most needed. Finally, compilers require costly instruction-set support to communicate with the microarchitecture that would likely result in additional fetch bandwidth being wasted.

The dynamic merge point predictor uses run-time information to find the nearest merge point. For example, consider the control-flow graph (CFG) shown in Figure 3.1. A compiler would identify block F as the merge point, because it is the only block guaranteed to execute after A. However, highly biased branches can effectively remove edges from the CFG. To illustrate this, Figure 3.1 uses dashed edges to identify branch directions that are rarely taken at run-time. If these edges are omitted, then block D becomes the merge point. Predicting block D as the merge point yields a merge point that is closer to the branch, but is sometimes inaccurate. Predicting block F is always correct, but is farther away than block D, making it less useful for performance.

Highly skewed branches prune edges of the CFG, producing merge points that are closer to branches. Our experiments show that 61% of conditional branches never change direction while an additional 9% of branches change direction <1% of the time [3]. The large number of highly biased branches suggests that identifying merge points at runtime will have a ma-

[3]Measured across the SPEC CPU2006 [3] benchmark suite

jor advantage over compile time.

3.2.2 Weaknesses of Prior Work in Merge Point Prediction

The previous state-of-the-art merge predictor proposed by Collins et al. [15] has several major weaknesses. First, their predictor is not a general solution. It is a collection of three heuristics that all rely on the compiler to generate code that fits into their model. As compilers change over time, their predictor may become less accurate. In contrast, our algorithm leverages branch mispredictions to find the place where the wrong path and the correct path overlap. We do not rely on compiler heuristics, which enables us to cover a lot more cases and achieve higher accuracy. In our experiments, we compare to the infinitely sized, unrealistic predictor proposed by Collins et al. Despite their model having an unrealistic storage budget, their model achieves an average accuracy of only 78% across the branch intensive workloads in SPEC 2006. Our realistic 4KB predictor achieves an accuracy of 95% on those same workloads.

These numbers do not match the numbers reported by Collins et al. In their paper [15], the authors report an accuracy of 95% for their infinitely sized predictor, however, our evaluation shows an accuracy of at most 78%. We have accounted for the discrepancy and attribute it to two factors. First, we do not account for branches with trivial merge points that are are unlikely to be mispredicted. Examples of this are constant length loop branches and function calls. In both cases, the merge point is trivial to predict, boosting the

accuracy of the merge predictor. However, in both cases the branch direction is also trivial to predict, meaning that there will likely not be a branch direction misprediction. If the branch predictor is correct, we will not make use of the predicted merge point, making the correct merge point prediction meaningless. We therefore do not count easy-to-predict loop branches and function calls as part of accuracy. In our system, only branches with low branch prediction confidence make use of the merge point predictor. Due to the high frequency of loop branches, removing them from consideration significantly lowers relevant accuracy. Second, we enforce all merge points identified by both predictors be points where control converges. Due to the methodology used in [15], some of the predicted values are not true merge points, but rather random intermediate places in the control flow graph. In our methodology, these points are counted as incorrect for both predictors.

3.3 Dynamic Merge Point Prediction

A merge point prediction consists of three parts: the PC of the merge point, the merge distance, and the independent register set. The merge distance is the predicted number of dynamic instructions in which the merge point is expected to be found. The predicted distance can be used to identify merge point mispredictions, and also serves to place an upper bound on the number of instructions between the branch and the merge point, which may be useful for some control independence strategies. The independent register set is the set of architectural registers that are predicted to be independent of the

branch. Post merge point instructions that source registers identified by the independent register set do not have any data-dependencies with instructions between the branch and its merge point.

3.3.1 Merge Predictor Design

The merge predictor design consists of three new structures: the Merge Point Predictor Table, the Update List, and the Wrong Path Buffer (WPB). Figure 3.2 shows a block diagram of all three structures. The WPB is responsible for detecting new merge points and installing them into the predictor table. The update list is responsible for tracking predicted entries and updating them appropriately.

Merge points are detected by observing both the wrong-path and correct-path of a branch. When a branch misprediction occurs, wrong-path instructions are copied from the Reorder Buffer (ROB) to the WPB. After the machine is flushed, each retired, correct-path instruction accesses the WPB. If there is a hit, then a new merge point has been found and is installed into the predictor table. Next time the branch is fetched, the predictor table supplies the merge point and an entry is allocated in the update list. When the branch retires, it activates its entry in the update list. Once activated, the update list entry monitors retiring instructions. If the predicted merge point retires within the merge distance without any unexpected register writes, then the prediction is correct, otherwise it is incorrect. In either case, the entry is updated and then removed from the update list.

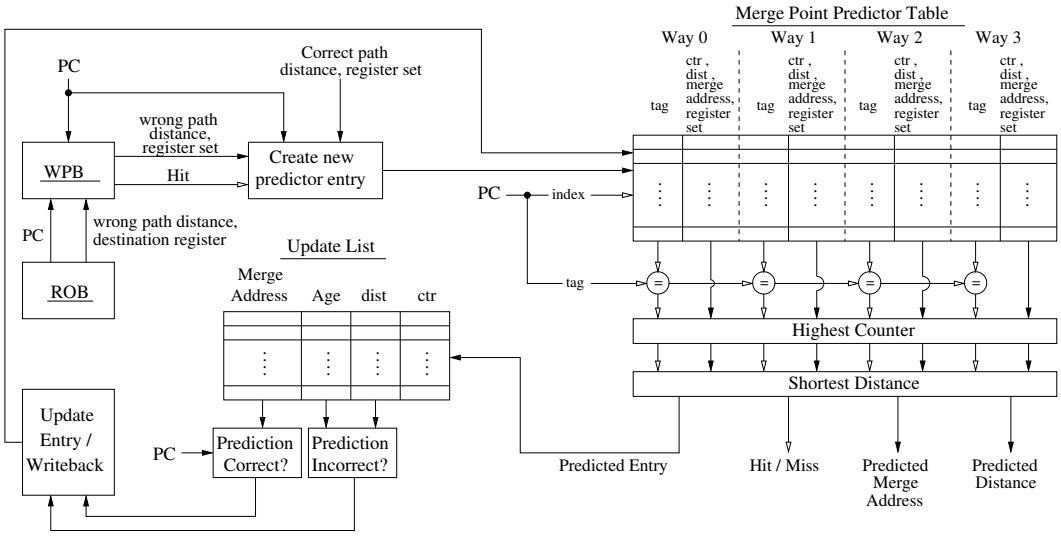

Figure 3.2: All three newly added structures: Merge Predictor Table, Update List and WPB.

Figure 3.2 shows the interactions between the three newly added structures and the ROB. The Predictor Table supplies the predicted entry to the Update List. The Update List compares entires to retired PCs until the merge point is confirmed or the merge distance is reached. At this point, the entry is updated and written back to the predictor. The Wrong Path Buffer (WPB) saves wrong-path PCs, supplied by the ROB, and compares them to correct-path PCs. When a match is found a new entry is installed in the predictor.

Section 3.3.1.1 discusses how new merge points are detected. Next, 3.3.1.2 discusses the implementation of the WPB. Section 3.3.1.3 discusses how predictions are made. Finally, section 3.3.1.4 discusses how the predictor is updated.

3.3.1.1 Detecting New Merge Points

Our design detects new merge points by exploiting branch mispredictions. Due to the large size of instruction windows and high fetch rates, it is common for processors to fetch many wrong-path instructions before detecting a misprediction. Our experiments show an average of 100 dynamic instructions fetched on the wrong path. Upon detecting a branch misprediction, wrong path instructions must be copied from the ROB into the WPB. Figure 3.3 shows an example. Instructions are copied from the ROB starting with the first instruction after the mispredicted branch, and ending upon one of three conditions: (1) there are no more instructions in the ROB, (2) the maximum merge distance is reached, or (3) another instance of the same branch is encountered in the ROB (i.e., a loop back to the branch). Instructions are copied from the ROB to the WPB by conducting a ROB-walk during the flush.[4] Each wrong-path instruction indexes the WPB with its PC and stores a wrong-path distance number and a bit-vector called the wrong-path independent register set. The distance number represents the number of dynamic instructions between the current instruction and the branch, while the independent register set represents the accumulated destination registers of each instruction up to this point. Finally, we tag the WPB with the PC of the mispredicting branch, and set a valid bit, indicating that the WPB should be compared to future

[4]We do not expect the ROB-walk latency to be an issue. It is not unusual for ROB-walks to be used during a flush to restore the state of the speculative register alias table. Furthermore, latency of ROB-walks are typically hidden by the front-end as it refills the pipeline.

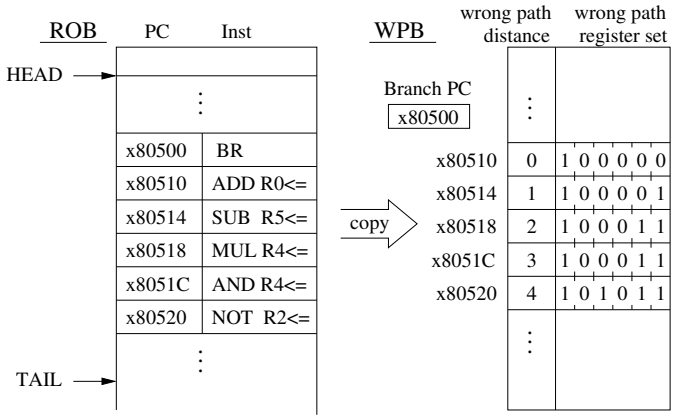

Figure 3.3: Wrong-path instructions copied from the ROB to the WPB

When the branch (PC=x80500) misprediction is detected, subsequent instructions are copied from the ROB to the WPB. The distance between the branch and each instruction is saved in the WPB. Additionally, the destination register of each instruction is saved in the the wrong-path register set bit-vector.

retired instructions.

After populating the WPB and flushing the machine, fetch is redirected down the correct path. When correct-path instructions retire, their program counters are used to index into all of the valid WPBs. Similar to filling the WPB, we continue until one of three conditions is met: (1) a PC hits in the WPB (2) the maximum merge distance has been reached or (3) the PC is equal to the PC of the mispredicted branch[5]. If there is a match (i.e., option 1), then we have found a merge point and install a new entry into the predictor table. If either option 2 or 3 occurs before finding a match, then we assume

[5]This happens when there is a loop back to the branch before encountering the merge point.

that there is no merge point and invalidate the WPB. Each WPB maintains a count of correct-path instructions that have indexed it called the correct path distance. Additionally, the WPB also tracks the correct-path independent register set by accumulating the destination registers of retired instructions into a bit vector.

Upon a WPB hit, the wrong-path distance and wrong-path independent register set are read from the WPB. The merge point is identified by the PC that hit in the WPB. The predicted distance is set to the larger of the wrong path distance and the correct path distance. Finally, the independent register set is formed by ORing the wrong-path bit vector and the correct-path bit vector. The entry is then installed into the predictor table and the WPB is invalidated.

3.3.1.2 Design of the WPB

Ideally the WPB would be a fully associative CAM; however, large CAMs are impractical. For that reason, we chose to implement the WPB as a 128-entry 4-way set associative cache. Organizing the WPB as a cache instead of a CAM creates the possibility for an entry to be evicted, creating false negatives. In our evaluation, we observed less than 1% false negative rate, which led to an almost negligible loss in coverage. The WPB uses the MRU replacement policy.

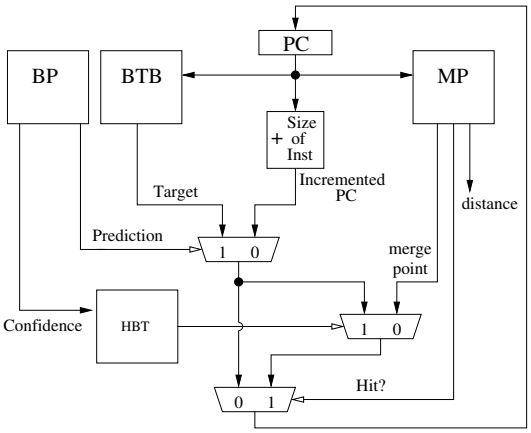

Figure 3.4: Interaction between the Branch Predictor (BP), Branch Target Buffer (BTB), and Merge Predictor (MP).

3.3.1.3 Making the Prediction

The PC is used to access the merge predictor in parallel with the branch target buffer (BTB) and the branch predictor. Figure 3.4 shows the connections between each of these structures. The entry supplied by the merge predictor is only considered if the Hard Branch Table (HBT) has identified the branch as hard-to-predict. The merge predictor is accessed as a typical set associative cache. If there is a miss, we defer back to the branch predictor.

If there is a hit, it is possible that multiple entries match the branch address. For example, consider the CFG in Figure 3.1. As discussed in section 3.2, the merge point of the branch in basic block A could either be D or F. It is possible that both D and F are detected by the WPB and are both installed into the predictor. In the event that two or more entries match in the predictor, the 3-bit saturating counter is examined and the entry with the

highest counter value is selected as the prediction. If two or more entries have equal counter values, then the merge entry containing the minimum merge distance is selected. We choose the entry with the minimum merge distance because predicting smaller distances results in smaller reservations in the instruction window. Once an entry has been selected for prediction, all entries that matched in the predictor are inserted into the Update List.

When new entries are installed into the predictor, it may be necessary to evict an older entry. Entries with the smallest counter value are the first victims for eviction. If all entries have equal counters, then the entry with the largest predicted distance value is selected as the victim.

3.3.1.4 Updating the Predictor

Once inserted into the update list, entries wait until the merge-predicted branch instruction reaches retire. At that point, the update list entry becomes active. An Update List entry contains the following information: (1) the PC of the merge-predicted branch, (2) a prediction age field, which is the number of dynamic instructions retired since the entry became active, and (3) the predictor table entry that will be updated and written back to the predictor. An entry remains in the Update List until either (1) a PC matching the merge address is found (meaning the prediction is correct), (2) the age field exceeds the merge distance (the prediction is incorrect), or (3) the PC of the

merge-predicted branch is seen retiring for a second time[6] (the prediction is incorrect). If the prediction is correct, the 3-bit saturating counter is incremented, otherwise the counter is decremented. Additionally, the destination registers of gap instructions are monitored. If any unexpected writes occur[7], then the prediction is considered incorrect and the machine is flushed.

We introduce another update policy called UPDATE_MAX. UPDATE_MAX will not remove an entry from the update list until the prediction age field has exceeded the max prediction distance. In this mode, all entries in the update list are treated as if their merge distances were equal to the max merge distance, regardless of the actual prediction. This allows the update list to detect if the merge address ever appears. If the merge address is encountered, then the 3-bit prediction counter is incremented and the merge distance field is set to equal the age field. This allows for the merge distance field to be increased as necessary. If the max merge distance is reached and the merge address is never found, then we decrement the 3-bit counter, as before.

Once the update policy has completed, the entry is removed from the Update List and written back to the predictor table. The Update List is a very small table with only 8 entries, and thus is organized as a fully associative cache.

[6]This happens when there is a loop back to the branch before encountering the merge point.

[7]Register writes not specified by the independent register set.

Table 3.1: System Configuration

1: Core	4-Wide Issue, 512-Entry ROB, 92-Entry Reservation Station, TAGE Branch Predictor [52], 3.2 GHz
2: L1 Caches	32 KB I-Cache, 32 KB D-Cache, 64 Byte Lines, 2 Ports, 3-Cycle Hit Latency, 8-Way, Write-Back.
3: L2 Cache	1MB 8-Way, 18-Cycle Latency, Write-Back.
4: Memory Controller	64-Entry Memory Queue.
5: Prefetchers	Stream: 64 Streams, Distance 16. Prefetch into Last Level Cache.
6: DRAM	DDR3
7: Merge Predictor Table	128 entiries, 4 way set associative, total size 1.6KB
8: WPB	128 entries, 4 way set associative, total size 1KB
9: Update List	8 entries, total size 113 bytes
10: Maximum Prediction Distance	100

3.4 Evaluation Methodology

To simulate our proposal, we use a cycle-accurate x86 simulator. The front-end of the simulator is based on Multi2Sim [61]. The simulator faithfully models core microarchitectural details and the cache hierarchy. Table 3.1 contains a list of microarchitectural details. Our simulator includes a 64KB TAGE [52] branch predictor configured similar to the version submitted to CBP 2014. We did not include the SC or L components of the TAGE predictor. We use the SPEC CPU2006 Integer benchmark suite [3] on the ref input set to evaluate our predictor. We use SimPoints [41] methodology to identify a single representative region, and run all of our benchmarks for 200 million instructions.

We use several metrics to evaluate the effectiveness of our merge point predictor. Accuracy is a measure of how often a prediction supplied by the merge predictor is correct. Coverage is similar to accuracy, however it factors in predictor misses (i.e., the merge predictor has no matching entry). Predicted

distance is the predicted number of dynamic instructions before encountering the merge point. True distance is the actual number of dynamic instructions seen between the branch and merge point. Finally, we compute branch misprediction coverage, which is the difference between old MPKI and new MPKI. New MPKI is calculated by adding the MPKI of the branch predictor and the MPKI of the merge point predictor.

We evaluate two versions of the merge point predictor, a version that uses the UPDATE_MAX policy and a version that does not. We will refer to them as MPPmax and MPP, respectively. Table 4.1 shows the complete predictor specification used in our experiments. We compare our predictor against the infinitely sized reconvergence predictor introduced by Collins et al. [15]. We will refer to their design as the reconvergence-inf.

3.5 Results and Analysis

Figure 3.5 (a) shows the accuracy of reconvergence-inf, MPP, and MPP-max respectively. The height of the bars indicates accuracy, while the stacks show predicted distances. Reconvergence-inf does not predict distance, so we have shown all of its predicted distances as the maximum distance. The final bar is the arithmetic mean (amean) of all workloads. Figure 3.5 (b) also shows prediction accuracy, but the stacks show true distance values. Figure 3.5 (c) shows the coverage results for each predictor.

Ideally, the predicted distance and the true distance would be equal, as some control independence strategies use distance to reserve space in the

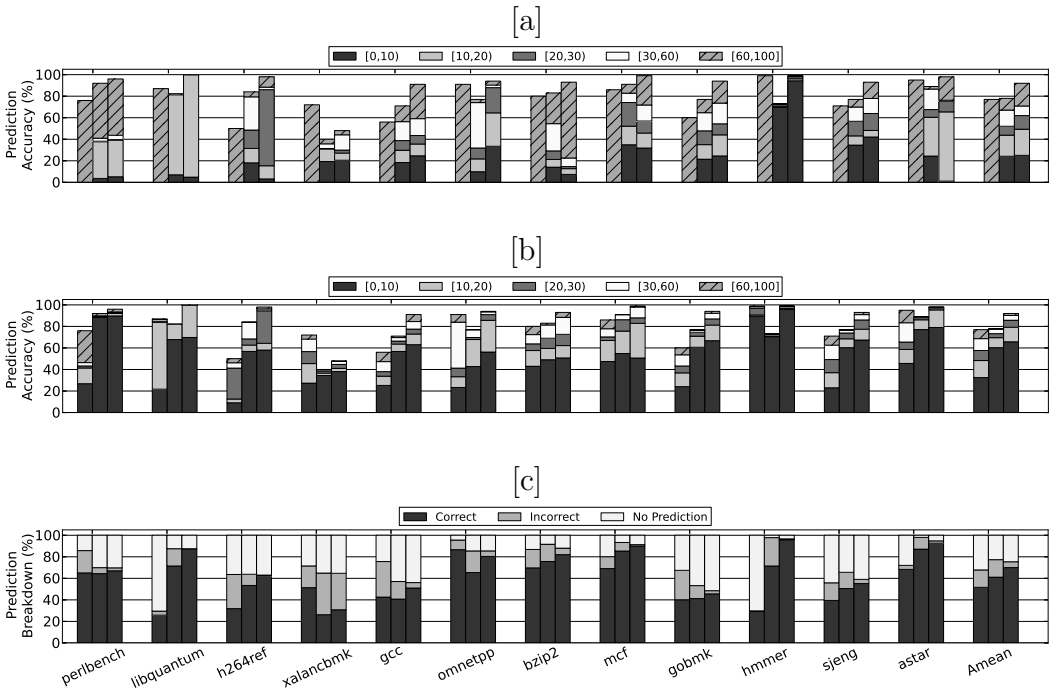

Figure 3.5: Final Merge Point Predictor Results
The bars, from left to right, represent the infinitely sized reconvergence-inf[15], MPP, and MPPmax. The top graph (a) shows prediction accuracy overlayed with predicted distance. The middle graph (b) shows prediction accuracy overlayed with true distance. The bottom graph (c) shows coverage.

instruction window. Unfortunately, this is not the case. Figure 3.6 shows the average difference between predicted and true distances for MPP and MPP-max. The height of each bar represents the wasted space in the instruction window due to overestimating the predicted distance.

Predicted distance can be overestimated for two reason. First, because the predicted distance is the larger of the correct-path distance and the wrong-

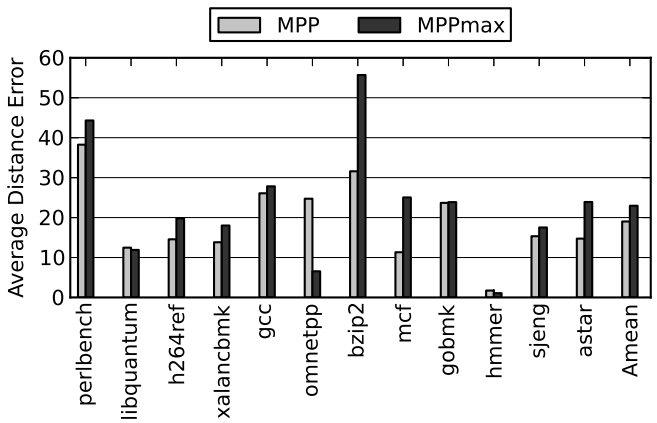

Figure 3.6: Average difference between predicted and true distances

path distance, it is possible that the smaller path was the one actually traversed at runtime, thus creating an error. Shortening the error in this case would be difficult, as the branch direction is not known. The second case is that the update policy is installing an unnecessarily large distance into the predictor.

The accuracy of MPPmax is higher than MPP in every benchmark, resulting in almost 14% higher accuracy on average. This is because the UPDATE_MAX policy strictly increases the predicted distance over time. Predicting larger distances can only increase prediction accuracy. However, the gain in accuracy comes at a cost. The negative effects of UPDATE_MAX are shown in Figure 3.6. MPPmax overestimates distance to a larger degree than MPP. This results in additional resources being wasted.

Both MPPmax and MPP outperform the infinitely-sized reconvergence-inf predictor. Collins et al. [15] work reports an accuracy of 95% for reconvergence-

inf, however, our evaluation shows an accuracy of at most 78%. We have accounted for the large discrepancy and attribute it to two factors. First, we consider predictions incorrect once the predicted distance has been exceeded. This is different from the methodology described by Collins et al., however this difference does not lead to a significant change in accuracy. Second, we enforce that all merge points identified by both predictors are points where control actually converges. Due to the methodology used by Collins et al., some of the predicted values are not merge points.

Accuracy numbers alone are not enough to understand the worth of a merge point predictor. It is important that we demonstrate that the merge point predictor is accurate when the branch predictor is not. Figure 3.7 shows the difference in MPKI between a branch predictor only design and a branch predictor + merge point predictor design. This represents the total number of mispredictions created by both the branch predictor and merge point predictor. The figure shows that MPPmax is able to replace a 56% of branch mispredictions with an accurate merge point prediction over a BP only design and a 51% improvement over reconvergence-inf. This significant improvement in branch misprediction coverage shows that MPPmax is highly accurate, even in the presence of hard to predict branches.

3.6 Prior Work

Control independence, as an alternative to branch prediction, was first proposed by Lam and Wilson [32]. While their study drew attention to the

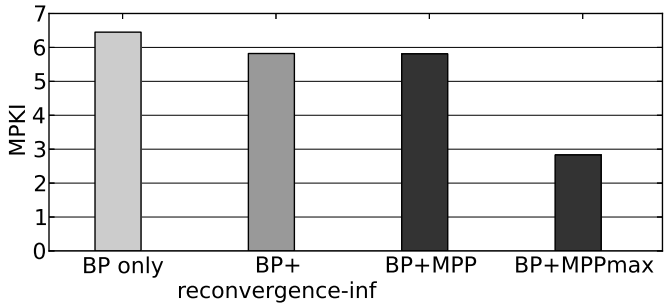

Figure 3.7: Projected MPKI with Merge Point Prediction

area, it made many assumptions that led to an inaccurate upper bound on performance [47, 60]. A more detailed analysis was done by Rotenberg et al. [47]. They devised six models designed to place an upper bound on the advantages of control independence. However, the focus of their work was limited to theoretical benefits of control independence, with no tangible solutions presented.

Skipper [13] was the first microarchitecture proposal for control independence that included a dynamic merge point predictor. Skipper identified hard-to-predict branches, then fetched post merge point instructions out-of-order to avoid prediction. However, Skipper is limited to only predicting if-then, if-then-else, and loops. Furthermore, it makes assumptions about the compiler and code layout, making it inaccurate and resistant to change. Additionally, Skipper only uses branch confidence to identify hard-to-predict branches. Our work also considers branch latency as an important factor.

Collins et al. proposed a reconvergence predictor [15], which attempted

to solve the limitations of Skipper. Their algorithm introduced 3 different heuristics for detecting merge points of if-then and if-then-else branches. Furthermore, they included support for call branches. However, their algorithm still depended on code layout, making it considerably less accurate. Additionally, their prediction mechanism provided no support for predicting distance or data-independence. Our predictor does not rely on the compiler, or make assumptions about code layout. It can identify merge points of all branches with sufficiently low distances and has a simply, extendable structure for tracking gap properties.

The SYRANT [42] work symmetrically allocated resources along both sides of a branch in preparation for a misprediction. Their merge point prediction mechanism also used wrong-path information, however, the paper focuses on the use case of their predictor, rather than evaluating the prediction mechanism itself, leaving the design of their predictor largely unknown. Additionally, their work is focused on optimizing branch mispredictions, while our work focuses on avoiding them altogether.

3.7 Future Work

History-based predictor. As discussed in this chapter, some branches have multiple dynamic merge points due to changes in branch behavior. In some cases, a branch may oscillate between more than one merge point each time it is encountered. Our current prediction mechanism will track these multiple merge points, however, it will predict the merge point which has been

occurring the most in the recent past. While this simple approach is accurate for most branches, a better design would use branch history as an index to predict which dynamic merge point is the most likely to be used next.

Improving the storage efficiency of the predictor table. Most commonly, the merge point of a branch is equal to its own target, as is the case of if-then branches [13]. Therefore, the target of the branch can serve as a base merge point prediction. This allows us to create a TAGE-like predictor design, where minimal storage is needed for branches whose merge point is equal to the target, and an tagged table is used to override the base prediction for cases where the merge point is not equal to the target.

Combining merge point prediction and target prediction. Merge point prediction is often used in place of branch direction prediction. Therefore, entries in the branch target predictor (IT-TAGE) can be used to store the merge point predictions themselves. This shared design would allow for an aggressive history indexed merge point predictor at minimal additional cost.

Chapter 4

Branch Runahead

4.1 Introduction

As branch predictors continue to improve, their accuracy on data-dependent branches remains roughly constant. Over time, data-dependent branches have become responsible for a larger share of the total remaining mispredictions. To illustrate this, Figure 4.1 shows the misprediction rate of the 32 most hard-to-predict branches from each benchmark. The left bar is the accuracy of a 64KB TAGE-SC-L [52], winner of the 2016 Champion Branch Prediction competition (CBP-2016) limited storage category, and the middle bar is the accuracy of MTAGE-SC [53], winner of CBP-2016 unlimited storage category. On average, MTAGE-SC is only able to reduce misprediction from 11% (TAGE-SC-L) to 9%, an improvement of only 18%. I propose using dependence chains— a short sequence of operations that can *pre-compute* the result of the branch before it is needed in the fetch stage.[1] Figure 4.1 also shows the accuracy of using dependence chains to pre-compute branch outcome (right bar). On average, dependence chains decrease mispredictions to 5%,[2] reducing misprediction by 55% (TAGE-SC-L) and 44% (MTAGE-SC).

[1]The ideas presented in this chapter were first published in [45], which is my own work.
[2]Some mispredictions still occur when the dependence chain diverges.

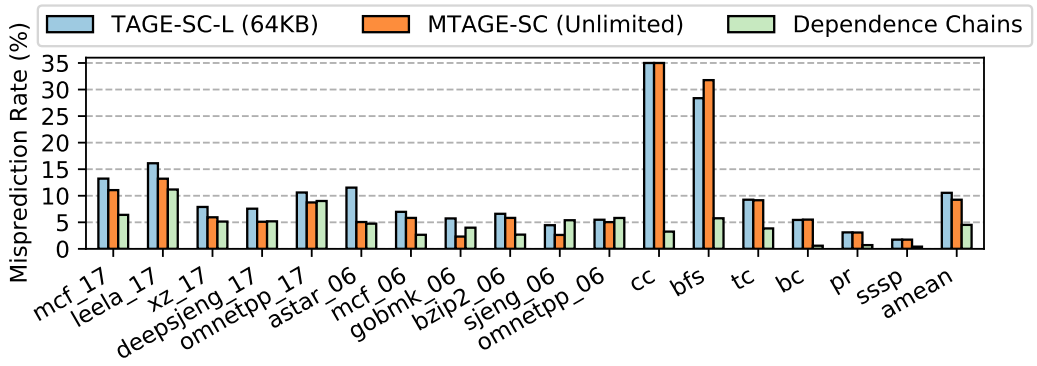

Figure 4.1: Misprediction Rate: TAGE-SC-L (64KB) vs MTAGE-SC (Unlimited) vs Dependence Chains for Hard-to-Predict Branches

This result clearly demonstrates that pre-computation can improve branch prediction for branches that TAGE fundamentally cannot predict accurately.

Unfortunately, prior work in pre-computation for branch prediction has primarily focused on heavy-weight, compile-time approaches. Here, the compiler creates a filtered version of the original program, only containing instructions necessary to compute the result of hard-to-predict branches. The filtered thread, or "helper" thread, is executed asynchronously on another core [59, 31], Simultaneous Multi-threading (SMT) context [64, 49, 11, 10], or on a dedicated unit within the core [54]. I argue these approaches are fundamentally more costly as they require re-executing most instructions in the program, and thereby require expensive resources to pre-compute the branch. Meanwhile, light-weight runtime approaches [10] are not able to run continuously, limiting their ability to provide timely predictions.

I propose Branch Runahead, a system that continuously executes lightweight

dependence chains to pre-compute the result of hard-to-predict, data-dependent branches. Dependence chains, which are extracted from the program at runtime, are far simpler than the helper threads proposed by prior work, which allows them to be accelerated using less hardware, rather than requiring another core or SMT context. In particular, Branch Runahead improves state-of-the-art in four key areas.

Light-weight Dependence Chain. The dependence chain is a short sequence of operations necessary to produce the result of a branch instruction. [3] Unlike helper threads, dependence chains are guaranteed to be simple. All dependence chains have fewer than 16 micro-operations, do not contain expensive operations such as integer divide or floating point operations, and do not contain any control flow instructions. Figure 4.2 shows that the average length of a dependence chain is fewer than 8 micro-operations. Branch Runahead dynamically filters the instruction stream to produce the exact sequence of operations needed to compute the branch outcome. The result is a light-weight sequence of instructions that can be accelerated efficiently.

Continuous Execution. Prior techniques using light-weight dependence chains [10, 23, 40] have struggled to execute continuously, as dependence chains do not contain control flow instructions. This causes them to diverge from the main thread quickly, reducing the accuracy of predictions. Branch Runahead solves this problem using our new merge point predictor. The merge

[3] More specifically, a dependence chain is the backwards dataflow slice needed to compute the branch.

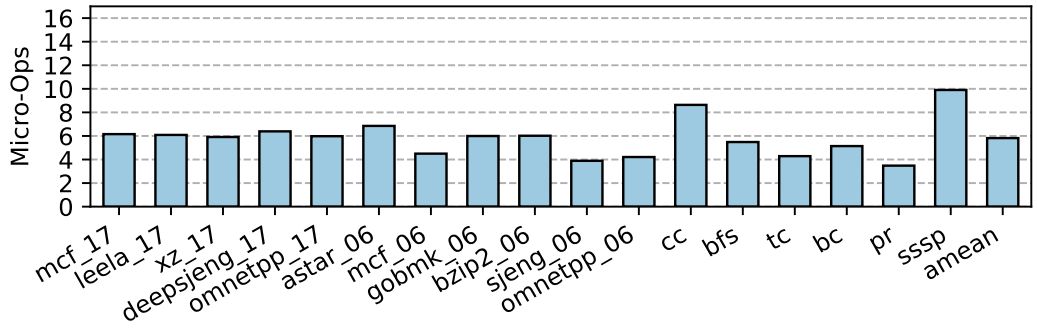

Figure 4.2: Average Length of Dependence Chains

point predictor detects control and data dependencies between branch instructions, which allows Branch Runahead to properly order dependence chains.

Timeliness. Pre-computation is effective only if the results are ready before the prediction is needed. Branch Runahead maximizes chain level parallelism by predicting the next dependence chain, allowing it to issue as early as possible.

Dependence Chain Engine. Branch Runahead executes dependence chains using a dedicated unit, the Dependence Chain Engine (DCE). I propose 3 variants of the DCE: an unlimited storage *Big* engine, a 17KB *Mini* engine, and a 9KB *Core-Only* engine, which shares reservation stations, physical registers, and functional units with the core. The Dependence Chain Engine consumes just 2.2% of the area of a typical out-of-order core (or only 1.4% in the case of the Core-Only model). As branch outcomes are produced, they are inserted into prediction queues, which override predictions supplied by the traditional branch predictor. Dependence chains begin executing in the DCE as

soon as their live-ins are known. Live-ins are copied from the physical register file during a branch misprediction and loaded into the DCE. This initializes the register file for the chain, effectively synchronizing it with the core. Dependence chains are not guaranteed to be correct, and occasionally diverge from the main thread. Once this is detected, the DCE is synchronized again by repeating the copy of the physical registers.

The contributions of this Branch Runahead are:

- To our knowledge, this paper is the first work to evaluate dynamically generated, light-weight dependence chains that are run continuously as a means to pre-compute the results of branch instructions.
- I demonstrate the importance of accurately identifying affector and guard dependencies between branches (section 4.4.4).
- I introduce a new method for merge point prediction, which is 95% accurate, compared to prior work which is only 78% accurate [44] (section 4.4.4).
- I introduce the Branch Runahead system, including dependence chain extraction, synchronization with the fetch unit, and the microarchitecture of the Dependence Chain Engine. We show that placing restrictions on chain extraction can guarantee the simplicity of the dependence chain, allowing it to be executed quickly and efficiently on the Dependence Chain Engine (DCE).
- I evaluate three methods for chain initiation, which promotes chain level parallelism and improves the timeliness of predictions.

In the remaining sections, I discuss the fundamental limitations of prior work, and how Branch Runahead addresses them. Then, I provide a detailed discussion of Branch Runahead, including implementation and important properties. Finally, I show that Branch Runahead, when configured under reasonable hardware constraints, reduces branch MPKI by 47.5% and increases IPC by an average of 16.9%.

4.2 Limitations of Prior Work

Branch Runahead is not the first to propose pre-computation as a substitute for branch prediction. In fact, many works have paved the way for Branch Runahead [63, 64, 49, 11, 10, 54, 31]. However, Branch Runahead is the first runtime only solution to execute light-weight dependence chains continuously. This allows Branch Runahead to execute further ahead with fewer hardware resources.

4.2.1 Limitations of Compiler-based Techniques

Most prior work relies on the compiler to identify candidate branches and extract helper threads [63, 64, 49, 11, 10, 54, 31]. Since compilers take a holistic view of the program, they can iterate over the control flow and data flow that lead up to a branch instruction and produce the exact minimum set of operations needed to compute the direction of the branch. Zilles et al. [63, 64] first observed, however, that building helper threads that computed branch outcome 100% accurately was not profitable, as it required too

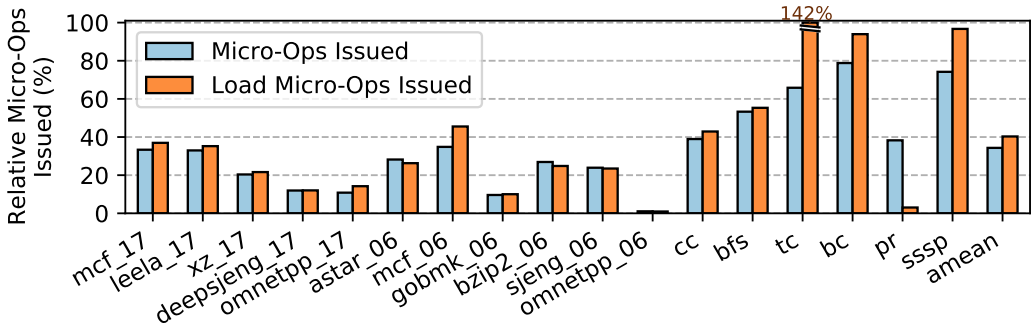

Figure 4.3: Increase in micro-ops due to Branch Runahead

many operations to be a part of the helper thread. Since then, research has focused on using profiling techniques during compile time to more aggressively remove instructions from the helper threads [49, 11, 31]. This results in a helper thread that is significantly simpler but no longer 100% accurate. Unfortunately, the effectiveness of these techniques relies heavily on the representativeness of the profiling data. Unrepresentative data can lead to inaccurate compiler optimizations that improperly reduce the helper threads, causing them to produce far less accurate predictions at runtime, increasing expensive synchronizations. In addition, introducing dependence chains at the compiler level requires changes to the Instruction Set Architecture (ISA) that chip makers are usually hesitant, if not unwilling, to make. In contrast, Branch Runahead requires no modifications to the ISA or compiler to achieve its full potential.

4.2.2 Limitations of Prior Runtime Techniques

SlipStream and its variants [59, 57] are runtime only techniques that pre-compute the direction of branch instructions. In SlipStream, two processors are used to execute a program. The A-stream runs a filtered version of the program ahead of the R-Stream. This enables the A-stream to communicate branch directions via a hardware queue between the two cores. Slipstream reports to remove an average of 15% of all retired instruction, leaving the remaining 85% in the A-stream as overhead.

In contrast, Branch Runahead extracts only the instructions needed to predict the targeted branch. As a result, dependence chains in Branch Runahead are far simpler than the A-stream. To illustrate this, Figure 4.3 shows the increase in micro-ops executed due to enabling Branch Runahead. On average, Branch Runahead executes 34.3% more micro-ops, significantly less than SlipStream's 85%.[4]

Difficult-path SSMT (DP-SSMT) [10] extracts dependence chains at runtime to pre-compute hard-to-predict branches. However, DP-SSMT requires a trigger instruction to begin each instance of the dependence chains. In contrast, Branch Runahead generates dependence chains that can execute continuously, as if they were in a loop. This allows Branch Runahead to run farther ahead than DP-SSMT. Furthermore, Branch Runahead considers

[4]The Slipstream numbers are values reported in [57], which used a slightly different set of benchmarks. However, there are several benchmarks that do overlap which confirm the large disparity.

affector and guard branches, which enable Branch Runahead to run ahead for longer intervals with high accuracy. DP-SSMT, on the other hand, generates dependence chains that only work if control goes down a predefined path.

Dependence Chains for Prefetching. Hashemi et al. propose using dependence chains for data prefetching [23, 22]. While both papers show the predictive power of dependence chains, Branch Runahead utilizes affector/guard branches and frequent synchronizations to improve the accuracy of the dependence chains. Carlson et al. [9] propose a new way of extracting dependence chains different from the method we use. Naithani et al. [40] use a similar technique, but instead of running chains on a separate pipeline, they issue chains during cycles where the core is idle. Both works rely on branch prediction to generate the correct dependence chain, making the techniques less useful as a branch prediction alternative.

4.2.3 Limitations of Heavy-weight Helper Threads.

Prior work requires helper threads to execute on another core [59, 31] or SMT context [64, 49, 11, 10] because helper threads still contain complex control flow that requires expensive out-of-order hardware to execute quickly. Requiring a separate core doubles the hardware cost for a single thread, and adds non-trivial latency for core-to-core communication. Requiring a separate SMT context requires all helper thread instructions be fetched, decoded, renamed, and access many other structures, like the Re-order Buffer (ROB), Load-Store Queue, etc.

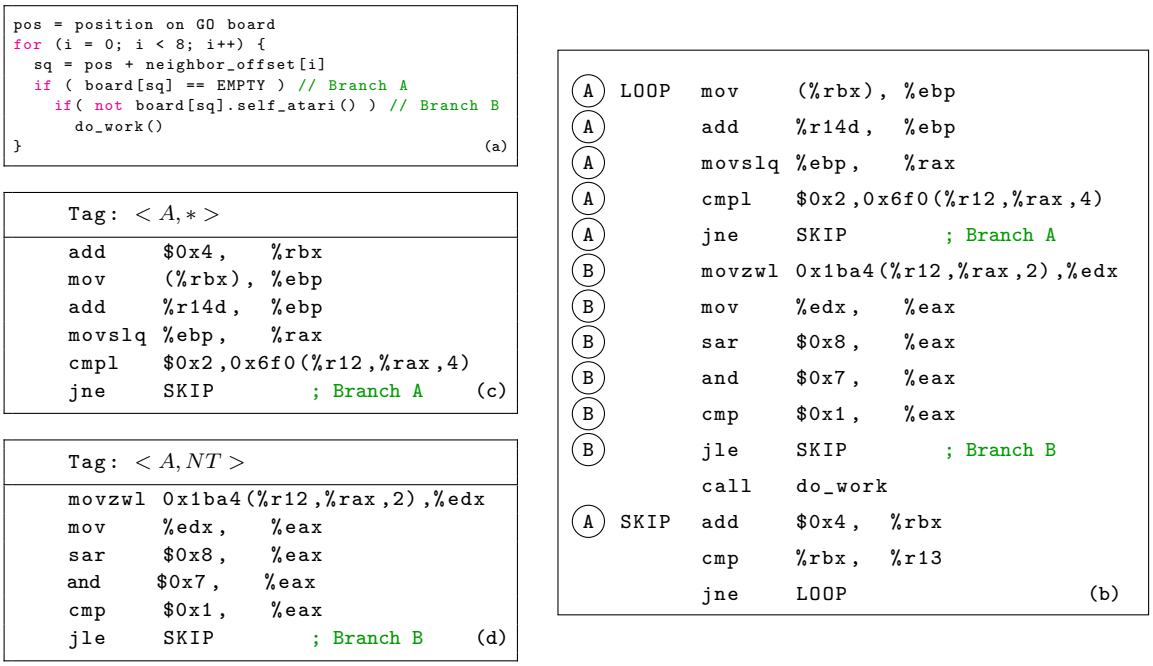

Figure 4.4: Code snippet from *leela*, a SPEC 2017 benchmark [3].

In contrast, Branch Runahead guarantees the simplicity of dependence chains. Dependence chains are stored in the Dependence Chain Cache as a sequence of micro-ops, so they do not need to be decoded. Further, most communication is internal to the dependence chain, allowing us to break Rename into 2 phases: a one-time local rename, and a dynamic global rename. This optimization also reduces the cost of reservation stations and physical registers. These simplifications motivate the creation of the Dependence Chain Engine (DCE): a dedicated unit for executing dependence chains.

4.3 Motivational Example

History-based branch predictors struggle with data-dependent branches because the branch outcome is not correlated to the branch history. Dependence chains, however, are a good fit for data-dependent branches because they use the application's own code to compute the direction of the branch.

Using dependence chains to predict branches. Figure 4.4a shows a code snippet taken from *leela*, a benchmark in the SPEC 2017 [3] benchmark suite. The code contains two hard-to-predict branches, A and B, which are two of the most frequently mispredicted branches in the benchmark. Branch A loads data from a random location on a GO board, then inspects it to see if the location is empty. The branch is hard to predict because it is a data-dependent branch with no correlated branches in the history.

Instead of trying to predict branch A, Branch Runahead extracts all instructions required to compute the outcome of the branch, forming the dependence chain for branch A. Figure 4.4b shows the assembly code generated by the compiler. Branch Runahead performs a backwards dataflow walk on branch A to find all instructions required to produce its outcome. These instructions (marked with the letter A) are included in the dependence chain for branch A. The resulting dependence chain is shown in Figure 4.4c.

Once extracted, dependence chains can be used to compute future predictions. The dependence chain is shipped to the Dependence Chain Engine where it is executed continuously, as if it were in a loop. This process generates

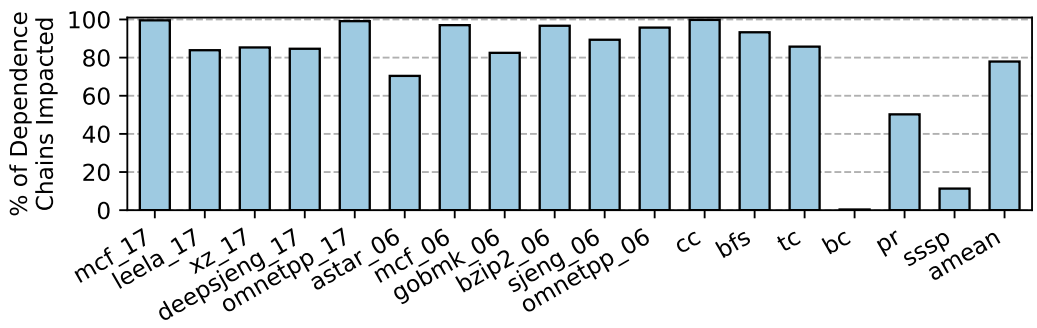

Figure 4.5: Dependence Chains with Affectors or Guards

accurate predictions for the duration of the **for** loop (until i reaches 8). Once that happens, the dependence chain diverges from the main thread, because it continues to assume that it is in a loop. This will cause future predictions for branch A to be inaccurate. Once this is detected, the core will synchronize with branch A's dependence chain and resume its execution.

Affector and Guard Branches. Frequent synchronizations inhibit Branch Runahead's ability to stay ahead of the main thread. Therefore, we need the dependence chains to remain accurate long enough for the Dependence Chain Engine to run ahead of the main thread. This requires Branch Runahead to be aware of frequently changing branches that may affect the instructions in the dependence chain.

Such branches are classified into two groups. *Guard branches* are branches that control the execution of another branch. For example, in Figure 4.4a, branch B is *guarded* by branch A and only occurs when branch A is not-taken. *Affector branches* are branches that can affect the source data

of other branches; i.e., the direction of one branch could have an impact on the data-dependencies of another branch. Both affector and guard branches can have a serious impact on the accuracy of dependence chains if ignored. Figure 4.5 shows the percentage of dependence chains in SPEC 2017 [3] that are impacted by affectors and guards.

As affector and guard branches impact such a large fraction of dependence chains, it is important that dependence chains be aware of affector and guard relationships. Branch Runahead uses our new dynamic merge point predictor to identify the merge point[5] of a branch at runtime. Once the merge point is known, it can be used to detect affector and guard relationships. This process will be discussed in detail in section 4.4.4.

What does the dependence chain for branch B look like? Similar to branch A, the chain extraction process starts by doing a backward dataflow walk starting at branch B. However, because branch A guards branch B, the chain extraction process terminates once branch A has been reached. Additionally, the dependence chain is tagged $< A, NT >$, representing the Program Counter (PC) of branch A, as well as the branch outcome (of branch A) required to execute branch B. The resulting dependence chain is shown in Figure 4.4d.

Tags are used to identify the action which initiates the execution of the dependence chain. For example, any time branch A is not-taken, that will

[5]The merge point of a branch is the instruction where control converges regardless of the true direction of the branch.

match with the tag $< A, NT >$ and the dependence chain for branch B can begin executing. Additionally, the dependence chain for branch A is tagged as $< A, * >$. The '*' denotes a wildcard, meaning that any outcome of branch A will match this tag and initiate branch A's dependence chain.

Biased branches and memory address aliasing.
Branch Runahead assumes that highly biased branches will remain biased and ignores them during chain extraction, even if a biased branch is also an affector or guard branch. Additionally, Branch Runahead assumes that memory address aliasing (i.e., overlapping memory addresses) between store-load pairs will also persist. These assumptions are, of course, not always true and can cause the dependence chains to diverge from the main thread. For example, once the **for** loop in Figure 4.4a terminates, the dependence chain for branch A will no longer be valid and any predictions it generates will likely be incorrect. Dependence chains will be deactivated when a misprediction is detected.

Putting it all together. Once the chains for branch A and B have been extracted, they are installed in a chain cache located in the Dependence Chain Engine. Newly installed chains cannot start executing until the core synchronizes the chain by initializing the chain's local register file with the correct input data. This does not happen until the next time either branch A or branch B mispredicts. When a misprediction occurs, the correct direction of the branch is broadcast to the Instruction Fetch Unit. Any chains whose tag matches the branch address and outcome are activated. At this point, newly

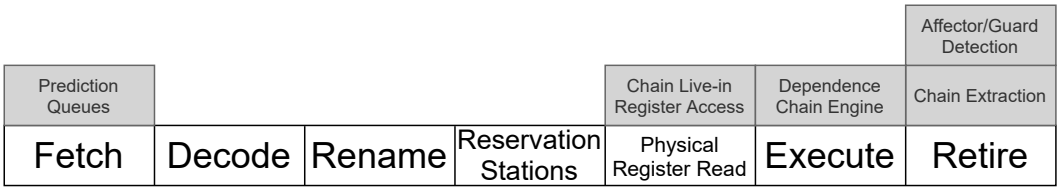

Figure 4.6: Pipeline Modifications

activated chains copy their live-ins from the physical register file and begin execution. Finally, when the dependence chain finishes execution, the branch address and outcome produced by the chain are used to identify the next set of chains. The dependence chains continue to execute, completely asynchronously from the core, until a misprediction from the dependence chains is detected. At that point, the mispredicting chain is synchronized by copying the correct values from the physical register file and chain execution resumes.

4.4 Branch Runahead Microarchitecture

Figure 4.6 shows a summary of the changes Branch Runahead requires on top of a typical Out-of-Order pipeline. We break these changes up into three categories: 1) dependence chain extraction, the process of identifying hard-to-predict branches and the uops that belong to their dependence chains. Once extracted, dependence chains are stored in the dependence chain cache. 2) Dependence Chain Control synchronizes the dependence chains with the core and allows them to execute continuously, producing near perfect branch predictions, which are used instead of predictions from the TAGE-SC-L predictor. Finally, the chains are executed on 3) the Dependence Chain Engine

(DCE), which is a specialized unit designed to execute dependence chains more efficiently than is possible on the core. Figure 4.7 shows a block diagram of the DCE. This section discusses Dependence Chain Control, followed by the microarchitecture of DCE, then concludes with dependence chain extraction and affector/guard detection.

4.4.1 Dependence Chain Control

Entering Runahead Mode. Once the dependence chains have been extracted, they are copied to the dependence chain cache where they wait to be *initiated*. Dependence chains cannot be initiated until their live-in data is synchronized with the core. Branch mispredictions present a convenient time to perform this synchronization, as the core backend and frontend are synchronized. When the core detects a branch misprediction, the branch address and outcome, which together form a tag, are used to look up an entry in the chain cache. If there is a hit, then the matching dependence chain is *initiated*; i.e., its uops are written into the reservation stations and its live-ins are copied from the core's physical register file. Additionally, the corresponding prediction queue is synchronized with instruction fetch. Once complete, the chain may begin execution.

Continuous Execution. Once initiated by the core, dependence chains execute continuously by using the dependence chain outcome to initiate future dependence chains. The aggressiveness of chain initiation impacts the level of chain level parallelism, which impacts the timeliness of the computed

branch outcomes. In this paper, we evaluate three initiation techniques.

Non-speculative Initiation. A dependence chain must finish execution entirely before initiating the next dependence chain. Once the dependence chain has finished execution, the pre-computed branch outcome and the branch address are used to index the chain cache and initiate all matching chains. Chain level parallelism is minimized in this mode as dependence chains must wait for their predecessors to finish execution before they can begin processing.

Independent-early Initiation. Chains with a wildcard tag are initiated when a triggering branch is issued into the reservation station. Recall from section 4.3 that some chains are marked with a wildcard tag, indicating that the direction of the triggering branch does not matter. In this case there is no need to wait for the predecessor chain to finish execution as the result of the branch will not affect whether or not the wildcard chain is initiated. Instead, wildcard chains are initiated as soon as their predecessor chains finish initiation. This mode increases chain level parallelism as chains with wildcard tags can now execute in parallel[6]. Non-wildcard chains, however, must wait for the predecessor chain to finish execution.

Predictive Initiation. Before, chains with non-wildcard tags required the direction of the triggering branch to be known in order to initiate them;

[6]Note that initiation simply affects when the chain is added to the reservation stations. The individual uops within the dependence chain are still required to wait until their data dependencies have been satisfied before they are scheduled to the functional unit.

however, in this mode the outcome of each branch is predicted at initiation time. This allows the branch address and the predicted outcome to index the chain cache and initiate non-wildcard matching chains early. If the prediction turns out to be incorrect, the speculatively initiated chains are simply flushed and the correct chains are initiated in their place. This mode maximizes chain level parallelism when the predicted branch outcomes are correct, or, in the case of mispredictions, results in chains being initiated no later than they would have in the prior two modes. In this mode, wildcard chains are handled exactly as they were in Independent-early Initiation. It is important to note that the prediction is only used to increase chain level parallelism; thus, any level of accuracy will likely improve the timeliness of branch outcomes. We use a simple per-branch 3-bit counter as the prediction mechanism.

Predictive Initiation, however, comes with the downside that dependence chains are now executing speculatively. This means that it is now possible for DCE resources to be wasting executing micro-ops that will eventually be flushed. This can waste energy, but also potentially create interference that slows down the DCE.

In all three modes, chain execution continues as long as the produced tags continue to hit in the chain cache. If ever the produced tag misses, then there are no more chains to initiate and runahead mode is exited.

Detecting Dependence Chain Divergence. Unfortunately, dependence chains will eventually diverge from the main thread. When this happens, the dependence chains will begin to produce incorrect predictions. Branch

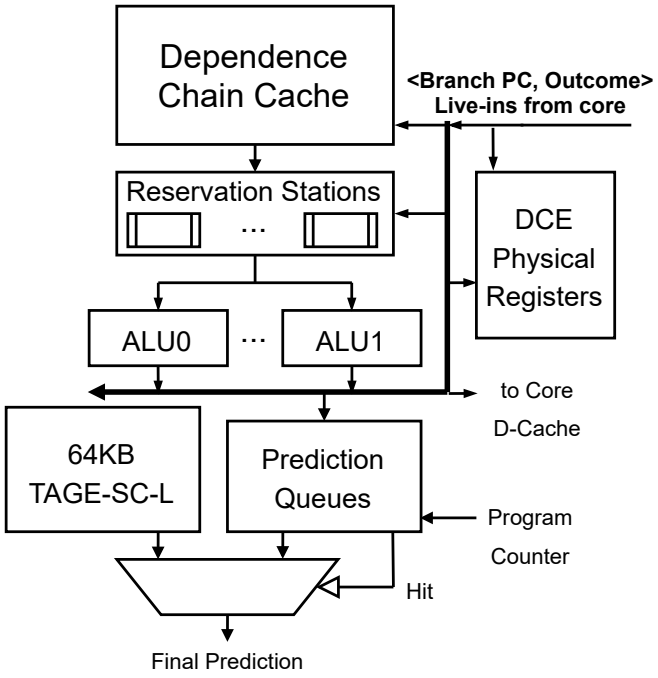

Figure 4.7: DCE Microarchitecture

Runahead monitors all prediction produced by dependence chains. Once a misprediction is detected, the chain is synchronized by once again copying the live-in values from the core to the DCE.

4.4.2 DCE Microarchitecture

The microarchitecture of the DCE is specialized to execute dependence chains more efficiently than the core. This is accomplished by observing that most of the communication between uops occurs within the dependence chain. Tailoring the microarchitecture of the DCE around this allows us to reduce the number of ports required for physical registers and reservation stations,

in turn reducing the cost-per-entry of those structures. Alternatively, we also evaluate a *Core-Only* implementation of the DCE, which shares physical registers, reservation stations, and functional units with the core. This section covers each component of the DCE microarchitecture, how it works, and what utility it provides towards improving branch prediction.

Dependence Chain Cache. Once the dependence chains have been extracted, they are copied to the dependence chain cache where they wait to be executed. The dependence chain cache holds up to 32 dependence chains and uses LRU as a replacement policy.

Rename and Instruction Scheduling. Instruction scheduling happens at two levels of granularity: global (i.e., initiating the dependence chains, discussed in section 4.4.1) and local (i.e., scheduling the uops within a chain). To accomplish this, Branch Runahead uses two levels of rename. Local rename happens once, during chain extraction, where communication within the chain is assigned a local physical register. Global rename happens dynamically, when the dependence chain is initiated, and results in assigning the dependence chain to a local register file and reservation station.

Physical Registers. The physical register file is divided into several local register files (Figure 4.8). Each local register file is treated as an independent, single-ported bank. Dependence chains read and write to their own local register file, except for live-in values [7], which are read from the local

[7]Values produced by another dependence chain or the core.

register file of the producer chain, creating the possibility of a bank conflict.

Reservation Stations. The reservation stations are divided into several local reservation stations, each with a capacity of 16 uops (1 chain). As uops are executed, their results are broadcast to the physical register file and to the reservation stations. Once all sources of a uop are ready, then the uop may be scheduled for execution. Uops are scheduled to execute out-of-order. We experimented with in-order instruction scheduling; however, we found that in-order execution was not able to expose enough Memory Level Parallelism (MLP) to significantly benefit the dependence chains.

Instruction Window. The physical register file and reservation stations together form the instruction window. The number of local register files and reservation stations directly affects the number of dynamic chain instances that are actively executing at once. Increasing the window size allows for more chain level parallelism; however, it can come at a significant cost. Alternatively, the entire instruction window can be shared with the core, minimizing cost, but also minimizing chain level parallelism. See section 4.5.2 for a detailed analysis.

Figure 4.8 shows an example. Here, a branch misprediction from the core triggers the tag $< A, * >$. The misprediction also triggers a synchronization between the DCE and the core. The chain's live-ins are read from the core physical registers and copied to a new local register file (red). While the live-ins are being copied, the matching dependence chain is issued into a reservation station and allocated a new local register file (blue). The chain's source

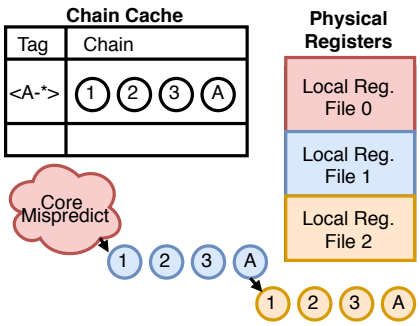

Figure 4.8: Global Rename Example

register file is set to red. Once the branch is issued, its address is broadcast to the chain cache to initiate any matching wildcard tags. In this example, the same chain matches, which triggers another dynamic instance of the chain to be issued (orange) and the chain's source registers are set to blue. Once the core has copied all live-ins to the red register file, the ready-bits in the physical register file will be set and the reservation station will be notified. At this point, the chain may begin executing. As uops execute, their results are written back to the appropriate local register file and the reservation station is notified.

Memory Accesses. The DCE shares the D-Cache and D-TLB with the core. The main thread is given priority to the D-Cache and D-TLB ports, and the DCE may only use these structures when available. Dependence chains do not contain any store instructions (see section 4.4.3), so the main thread does not have to worry about data corruption by the dependence chains.

The Prediction Queues. The prediction queues ensure that predictions between the DCE and core are synchronized. The size of each prediction queue also limits how far ahead (or behind) the DCE can be, which can affect performance. The DCE contains 16 per-branch prediction queues. When a dependence chain is initiated, a slot in the corresponding prediction queue is allocated. [8] Finally, when the dependence chain finishes execution, it pushes the outcome of the branch into the prediction queue. When the branch is fetched, it will see that its queue contains a prediction and will use that result instead of TAGE-SC-L. However, if the core fetches the branch before the DCE has computed the next prediction, then the slot is marked as consumed, even though it has not yet been filled. Later, when the DCE finishes computing the prediction, the already consumed slot in the prediction queue will be filled in case there is a recovery. To maintain the state of the queue, we use three pointers: *DCE push* for inserting new predictions, *core fetch* for consuming predictions at fetch, and *core retire* for removing retired predictions.

Recovery. During a branch recovery, the core fetch pointer is restored to its state before the misprediction, effectively reinserting previously consumed predictions into their original positions in the queue. This is accomplished by checkpointing the state of the core fetch pointer at each branch.

Prediction Throttling. Each prediction queue has a 2-bit throttle counter, which is incremented when the DCE is correct and TAGE is incorrect

[8] Slots must be allocated at initiation to ensure they appear in the prediction queue in program order.

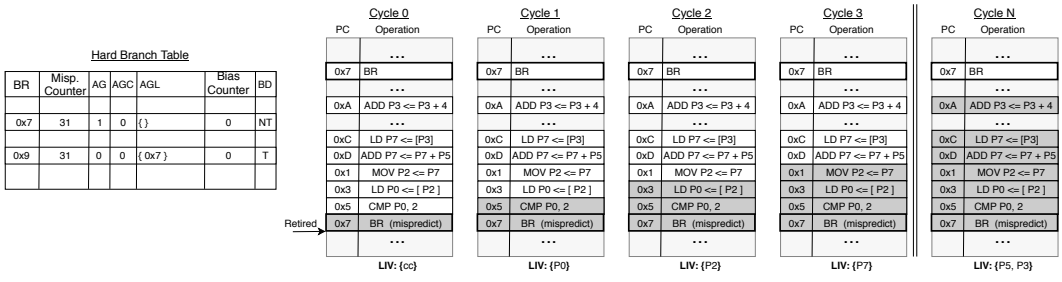

Figure 4.9: The Chain Extraction Buffer (CEB)

and decremented when the DCE is incorrect and TAGE is correct. When the counter is negative, predictions produced by the DCE are ignored.

4.4.3 Chain Extraction Hardware

Detecting Hard to Predict Branches. The *Hard Branch Table (HBT)* (Figure 4.9), introduced in Chapter 2, detects hard-to-predict branches. New entries are allocated when a conditional branch retires (if space available). Each entry consists of a 5-bit saturating misprediction counter that is incremented upon retiring a mispredicted branch. A branch is considered hard-to-predict when its misprediction counter saturates. Misprediction counters are periodically decremented by 15 every 1000 retired branches. Old entries can be overwritten when their counter is 0.

Tracking Affector and Guard Branches. In addition to identifying hard-to-predict branches, the HBT also keeps track of affectors and guards. Once detected (section 4.4.4), affector/guard branches are allocated in the HBT. The *AG* field is set to indicate the branch is an affector/guard, which

allows the branch to remain in the table even if it is not hard-to-predict. Affector/guard branches can only be replaced when the hard-to-predict branch they are associated with is removed. In addition, the affector/guard branch is added to the *affector/guard list (AGL)* field of the hard-to-predict branch.[9] If this branch was not previously in the affector/guard list, then the *affector/-guard changed (AGC)* field is set, indicating that a new affector/guard branch was found.

Branch Runahead ignores highly biased affector/guard branches. Therefore, the HBT tracks the bias of each affector/guard branch using a 7-bit bias counter, which is incremented when the direction of a retired branch matches the direction stored in the *Biased Direction (BD)* field and is decremented by 9 periodically[10]. If an affector/guard branch is found to be biased, then it is removed from the hard-to-predict branch's affector/guard list, and the AGC field is set if appropriate.

Extracting the Dependence Chain. The chain extraction algorithm is adapted from Hashemi et al. [23] where the authors use dependence chains to create prefetches for load instructions. Chain extraction begins when a hard-to-predict branch is retired.[11] Chain extraction terminates when either 1) a second instance of the same branch is found, or 2) an affector/guard

[9]The AGL is stored as a bit-vector, 1 bit per entry in the HBT.

[10]The decrement amount and counter width was calculated using an arithmetic model that theoretically detects a bias of 90% or more with a false positive rate of 1%.

[11]The branch must be contained in the HBT and have either saturated its misprediction counter or be randomly selected. Branches are randomly selected with a 1% probability.

branch is found. Upon termination, the dependence chain is tagged with the PC and outcome of the terminating branch and installed into the dependence chain cache.

Chain extraction performs a backwards dataflow walk starting at the most recently retired hard-to-predict branch instruction. To facilitate this, we add a circular buffer, called the chain extraction buffer (CEB), that holds the last 512 micro-operations (uops) retired. Uops in the CEB are searched cycle-by-cycle to see if they belong to the dependence chain. Figure 4.9 shows an example. In cycle 0, the second, younger dynamic instance of the mispredicting branch (shaded in grey) is added to the dependence chain and the search list (LIV) is initialized with the live-in table entry associated with this branch. Additionally, all of the branch's source registers (i.e., the condition code register) are added to the search list. Then, the CEB is iteratively scanned for uops whose destination register(s) match in the search list. On a match, we 1) add the matching uop to the dependence chain, 2) remove the matching register(s) from the search list, and 3) add the source registers of the matching uop to the search list. In cycle 1, the CEB is scanned for uops that write the condition codes, resulting in the addition of the CMP uop to the dependence chain and its source registers added to the search list. In cycle 2, the CEB is scanned for uops that write to R0. This results in the LD uop being added to the dependence chain, P0 being removed from the search list, and the sources of the LD (P2) added to the search list. When a load op is found, its address is compared to other addresses in the CEB store buffer. If a corresponding

store op is found, then the store is also added to the dependence chain. In cycle 3, the MOV uop is added to the dependence chain because of its write to P2. Finally, in cycle N, we reached the initial instance of the branch, ending chain extraction. The shaded uops make up the final resulting dependence chain. Several uops in the CEB were omitted because they are not added to the dependence chain.

Chain extraction takes place one chain at a time, off the critical path, and is not latency sensitive. The process takes multiple cycles, as shown in Figure 9. In all of our experiments, we model this latency accurately [12]; however, we experimented with much longer latency (1000s of cycles) and found no sensitivity. This is because chain extraction very rarely produces a chain that is not currently in the chain cache.

Live-in and Live-out Tables. The live-in and live-out tables hold the architectural live-ins/outs for each chain detected during chain extraction. In Figure 9, the architectural registers corresponding to the live-in vector (LIV) are stored in the live-in table at the end of chain extraction. Further, the Live-out vector, i.e., architectural registers corresponding to P0, P2, P3, P7, is written into the live-out table. This information is used during local rename and during synchronizations with the core.

Local Rename. Local rename happens once during chain extraction. During this process local registers (i.e., communication within a chain) are

[12] uops in CEB / retire width

renamed and global registers (communication between chains) are identified and partially renamed to prepare for global rename. Local registers are renamed to minimize physical register footprint. Global registers are identified by comparing the live-in/live-out registers of the chain with the live-out/live-in registers of the producer/consumer branches, respectively. Global registers are renamed in-order to guarantee the same name is used between different chain extractions.

Dependence Chain Optimizations. All move uops are move-eliminated. Further, because store-load pairs detected during chain extraction are logically equivalent to a move, all store-load pairs are move eliminated as well. This optimization guarantees that dependence chains will not contain any store instructions.

4.4.4 Detecting Affector and Guard Branches

Branch Runahead uses the merge point prediction algorithm discussed in Chapter 3 to detect control and data dependencies between dependence chains. Here, the WPB is used to detect the merge point of a target branch. The WPB is extended to then detect Affector and Guard Branches.

Detecting Guard Branches By definition, a branch guards any branches that are observed between itself and the merge point (excluding biased branches). Therefore, any branches observed on the wrong-path or correct-path during merge point prediction are marked as being guarded by the merge predicted branch. If no merge point is found, then no branches are

marked.

Detecting Affector Branches The merge predicted branch is an affector for any branch that sources data that is affected by the direction of the merge predicted branch. Fortunately, the merge point predictor provides us with the both-path dest set, which marks all registers/memory addresses affected by the merge predicted branch. To detect affectee branches, we use the both-path dest set and the poison algorithm adapted from Runahead Execution [39]. Registers and memory addresses marked in the both-path dest set are initialized as poisoned. As correct path instructions after the merge point retire, they propagate any poison they source to the destination register. If an instruction outputs to a poisoned register without sourcing any poisoned registers, then the destination register poison is removed. Any branch, including the merge predicted branch, that sources poison is considered to be an affectee branch [13] (i.e., the merge predicted branch is an affector of the poison sourcing branch). The process terminates when the second instance of the merge predicted branch is seen or when the maximum merge point distance has been reached.

[13] Excluding biased branches.

Core	4-Wide Issue, 256-Entry ROB, 92-Entry Reservation Station, 3.2 GHz, 64KB TAGE-SC-L Branch Predictor [52]. Modeled by Scarab [2].
WPB	128-entry, 4-way, max merge point distance 256 uops.
L1 Caches	32 KB I-Cache, 32 KB D-Cache, 64 Byte Lines, 2 Ports, 3-Cycle Hit Latency, 8-Way, Write-Back.
L2 Cache	2 MB 12-Way, 18-Cycle Latency, Write-Back.
Memory Controller	64-Entry Memory Queue.
Prefetchers	Stream: 64 Streams, Distance 16. Prefetch into LLC.
DRAM	DDR4, 8Gb, x8, 2400R, Modeled by Ramulator [30].

Table 4.1: Baseline Configuration

	Core-Only (9KB)	**Mini (17KB)**	**Big (Unlimited)**
uOps	Integer: add/multiply/subtract/mov/load. Logical:and/or/xor/not/shift/sign-extend.		
Chain Cache	32-entry (2KB) 1 uop per entry, 4B per uop.		1024-entry
PRF	0 (0KB)	64x 8-entry (4KB)	1024x 8-entry
RSV	0 (0KB)	64x 32-entry (4KB)	1024x 32-entry
MSHRs	48-entry	48-entry	64-entry
Prediction Queue	16x 256-entry (4KB)		1024-entry
HBT	64-entry (1KB)		
CEB	512-entry (2KB)		2048-entry

Table 4.2: Branch Runahead Configuration

4.5 Results

4.5.1 Evaluation Methodology

To simulate our proposal, we use Scarab [2]— an open source simulator commissioned by Intel in their Intel/NSF FoMR initiative [1]. Scarab is an execution-driven, cycle-accurate x86 simulator whose front-end is based on PIN [34]. The simulator faithfully models core microarchitectural details, the cache hierarchy, wrong-path execution, and includes a detailed non-uniform access latency DDR4 memory system, modeled by Ramulator [30]. We model the 64KB TAGE-SC-L [52] branch predictor with the configuration submit-

ted to CBP-2016. The 64KB TAGE-SC-L is the best known realistic branch predictor. Table 4.1 describes our system.

Branch Runahead Configuration. Branch Runahead is evaluated on three configurations— Core-Only (9KB), Mini (17KB), and Big (unlimited). Table 4.2 contains the details of each configuration.

Benchmarks. We evaluate Branch Runahead on SPEC CPU2017 Integer Speed, SPEC CPU2006 Integer [3] and GAP Benchmark Suites [8]. From that set, we select the branch misprediction intensive benchmarks with an average MPKI greater than 2. We use the SimPoints [41] methodology to identify anywhere between one to five representative regions per benchmark. We run each region for 200 million instructions, then compute the weighted average of all the regions. We run SPEC benchmarks on the ref input set, and use *-g 19 -n 300* inputs for GAP. If there is more than one ref input, then the benchmark is run on each input, and a weighted average, weighing each input by the total dynamic instruction count, is used to compute a single metric for the entire benchmark.

Energy and Area. We model chip energy and area using McPAT [33]. The DCE is modeled as a stripped down core, removing structures like decode, register rename, floating point pipeline, prefetchers, and others required for maintaining precise state, such as the ROB.

Metrics. We use Instructions Per Cycle (IPC) as the performance metric and Branch Mispredictions Per Kilo Instruction (MPKI) to evaluate

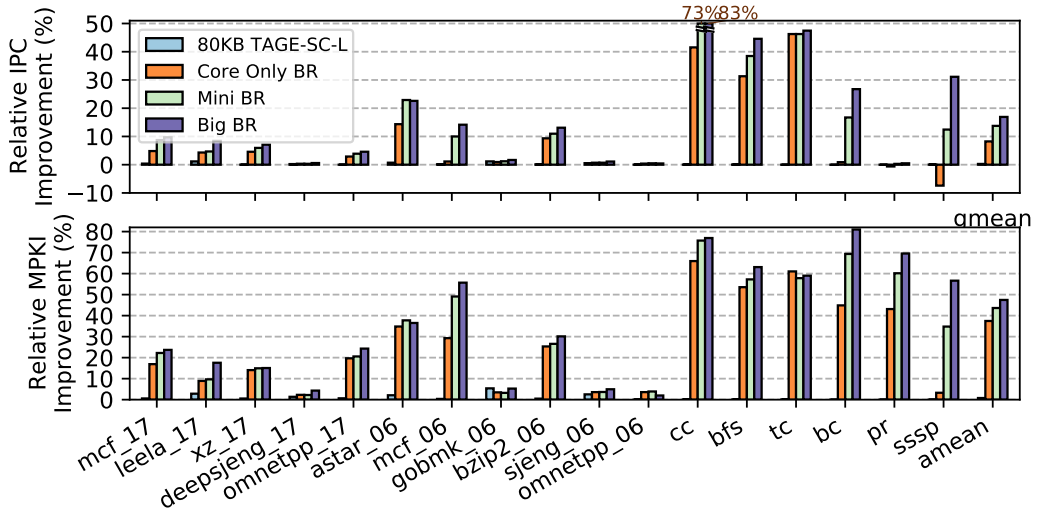

Figure 4.10: IPC and MPKI Improvement of Branch Runahead compared to 64KB TAGE-SC-L

improvements in prediction accuracy. MPKI Improvement is computed as the difference between Branch Runahead MPKI and TAGE-SC-L MPKI, normalized to TAGE-SC-L MPKI.

4.5.2 Branch Runahead Results

Figure 4.10 shows the performance results for the Core-Only, Mini, and Big implementations of Branch Runahead. The results show that Branch Runahead reduces MPKI by an average of 37.5%, 43.6%, and 47.5% and increases IPC by an average of 8.2%, 13.7%, and 16.9%, respectively. The trade-off comes down to cost vs chain level parallelism. Big-Branch Runahead maximizes chain level parallelism by providing the most physical registers and reservation station entries, while the Core-only model minimizes these same

qualities.

Figure 4.10 also shows the performance of an 80KB TAGE-SC-L predictor (left most bar), which requires roughly the same storage overhead as Mini Branch Runahead (16KB). However, the 80KB TAGE-SC-L only improves MPKI by 0.8%, resulting in only 0.3% improvement in IPC. Mini Branch Runahead improves the misprediction rates of targeted branches by an average of 55%, while 80KB TAGE-SC-L has a negligible effect on these same branches. This result supports the claim that history-based predictors are fundamentally unable to predict these data-dependent branches.

Limits of Branch Runahead. Big Branch Runahead uses unlimited storage to demonstrate the maximum potential of Branch Runahead. In this model, parameters of the microarchitecture are increased far beyond their reasonable limits. As Figure 4.10 shows, Big Branch Runahead improves MPKI over Mini Branch Runahead by only 3.8%, suggesting that Mini Branch Runahead is very close to its peak potential.

Limits of History-based Predictors. Figure 4.11 (top) compares Big-Branch Runahead to MTAGE-SC, winner of the unlimited storage category in CPB-2016 [53]. While MTAGE-SC improves MPKI significantly, particularly in the SPEC workloads, it is still outperformed on average by Big Branch Runahead (middle bar). This is because MTAGE-SC performs poorly on GAP workloads, which are dominated by data-dependent branches. Combining MTAGE-SC and Big Branch Runahead (right bar) further improves MPKI on every benchmark, demonstrating that Branch Runahead is capable

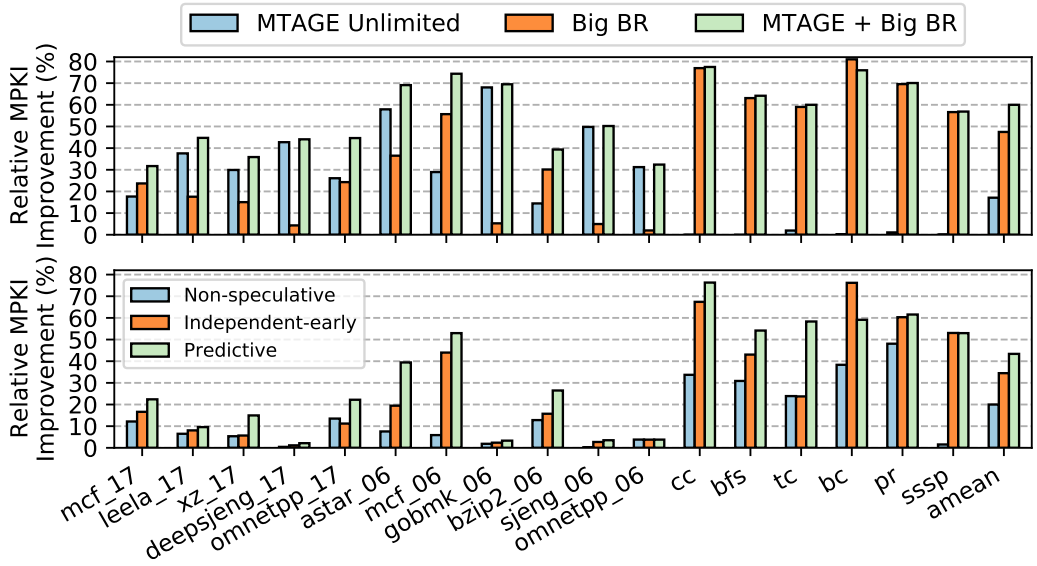

Figure 4.11: MPKI Improvement of MTAGE and Branch Runahead (top) and MPKI Improvement of Chain Initiation Methods (bottom)

of predicting branches that TAGE fundamentally cannot predict.

Chain Initiation Method. Chain Initiation is the most important factor towards improving timeliness. As discussed in section 4.4.1, Chain Initiation affects chain level parallelism. Figure 4.11 (bottom) shows the MPKI improvement of each initiation method. Unsurprisingly, the Predictive Initiation method, which maximizes chain level parallelism, has the highest impact on MPKI. While this method does produce the highest degree of performance, it comes at the cost of flushing the DCE on a misprediction, which wastes energy.

Timeliness of predictions. DCE predictions need to be timely in

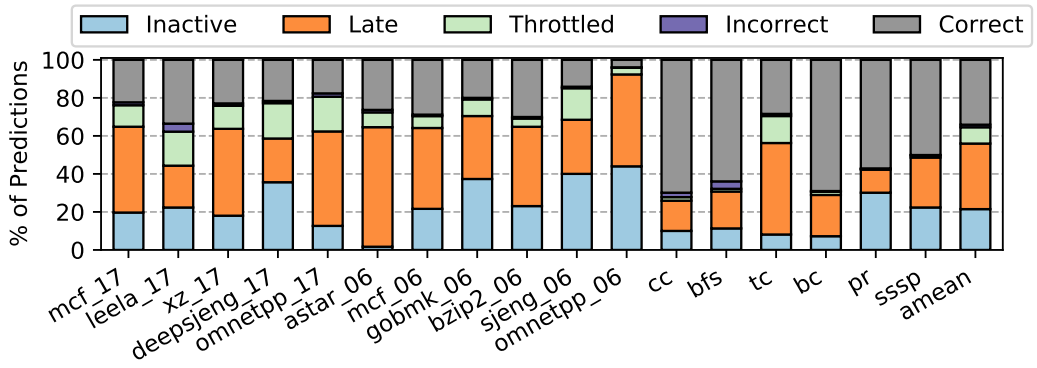

Figure 4.12: Prediction Breakdown

order to be useful. The stacked bars in Figure 4.12 show the fraction of predictions supplied by the DCE that are inactive or late. The inactive category means that, at the time the core needed the prediction, no dependence chains had been activated to produce that prediction. This generally happens once the branch is fetched, but before the first synchronizing misprediction occurs. Branch Runahead requires a mispredicted branch to synchronize, which unfortunately has the effect of activating chains late. The late category refers to predictions which have active chains, but are generated too late to be useful for the core. This generally happens when the dependence chain contains too many long latency operations. The throttle category refers to predictions that are throttled (as described in section 4.4.2). The remaining two categories, correct and incorrect, refer to predictions that are used by the core. This figure demonstrates two things. First, predictions generated by Branch Runahead are very accurate, with nearly all of the used predictions being correct. Second, nearly 40% of predictions are generated on time. Timeliness is the

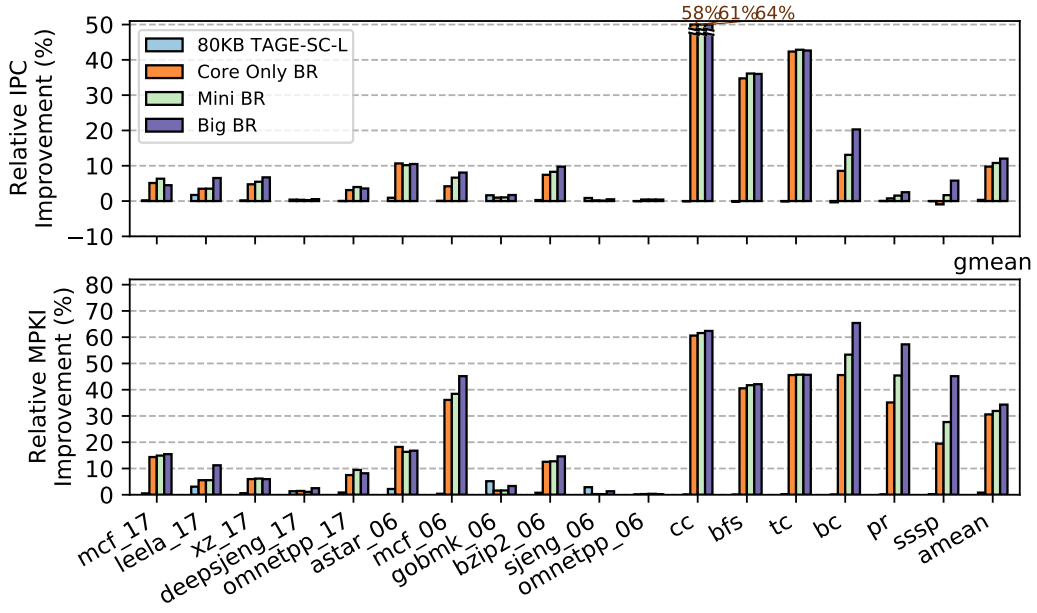

Figure 4.13: IPC and MPKI Improvement of Branch Runahead on a 16-wide fetch, 1024-entry instruction window baseline.

most difficult issue Branch Runahead faces, with late predictions making up the largest category outside of correct predictions.

Branch Runahead on a larger baseline machine. Branch Runahead achieves improvement by delivering accurate predictions to the frontend before the branch is fetched. Therefore, it is fair to ask if Branch Runahead will continue to achieve performance as the fetch rate of the baseline machine is increased. Increasing the fetch rate of the baseline machine puts pressure on Branch Runahead to deliver the branch results more quickly; however, it also increases the cost of each branch misprediction, thereby making Branch Runahead more valuable. Figure 4.13 shows the results of running Branch

Runahead on top of a machine that fetches 16 instructions per cycle and has a Re-order Buffer and Reservation Stations capable of holding 1024 instructions. The figure shows that the performance improvements of Branch Runahead decreases slightly across all workloads. This is primarily due to the stricter timeliness constraints placed on Branch Runahead. Additionally, the performance difference between the Core-only, Mini, and Big models also decreases. This is partly due to the fact that Core-only performance increases slightly due to a larger availability of resources in the larger instruction window.

Why does Mini Branch Runahead out perform Big Branch Runahead on some workloads? The results in figure 4.10 show that Mini Branch Runahead achieves a higher performance and a higher MPKI than Big Branch Runahead. This happens when the the Mini-DCE is able to achieve a higher performance than the Big-DCE. This is typically caused by one of two factors. First, the big-DCE generates a higher degree of cache pollution, due to the fact that it more quickly executes dependence chains. When the dependence chains inevitable diverge from the main thread, the memory accesses that they generate are often not useful to the main thread nor to future dependence chains. In extreme cases, this pollution can slow down both the main thread and future dependence chains. Second, the big-DCE can generate more interference for the main thread by virtue of creating more memory accesses in flight. These memory accesses create more traffic that can slow down either the main thread or other dependence chains, thereby slowing down Branch Runahead.

Impact on Security. In light of the recent Spectre and Meltdown attacks, more attention than ever is placed on ensuring new features to not unlock new attacks. Branch Runahead extracts dependence chains and executes them continuously, as if they were in a loop, to generate near perfect branch predictions. However, at some point these dependence chains will diverge from the main thread. Upon detection, the dependence chains are synchronized with the main thread and restarted; however, this is not before several wrong-path memory accesses are sent to the memory system on behalf of Branch Runahead. It is important to note that, just as in regular execution, these wrong path memory accesses would be manipulated to access private data. Therefore it is important that permissions are checked *before* the memory access is initiated. Branch Runahead accesses the D-TLB before all memory accesses, just as in the baseline system. Further, because the dependence chains are generated from the applications own code they do not create any new attack vectors that would not be available to an attacker already. Dependence chains only make assumptions about the code in two cases: 1) biased branches and 2) memory disambiguation. However, these two cases are both predicted in the baseline machine as well. For example, Branch Runahead assumes that biased branches will remain bias. However, this is not unlike any branch predictor that would make the same assumption.

Sweeps. Figure 4.14 shows MPKI improvement for a DCE with various configurations [14]. Parameters are swept up to the values used for the Big

[14]Due to the large number of simulations, sweeps ran for 10 Million instructions, as

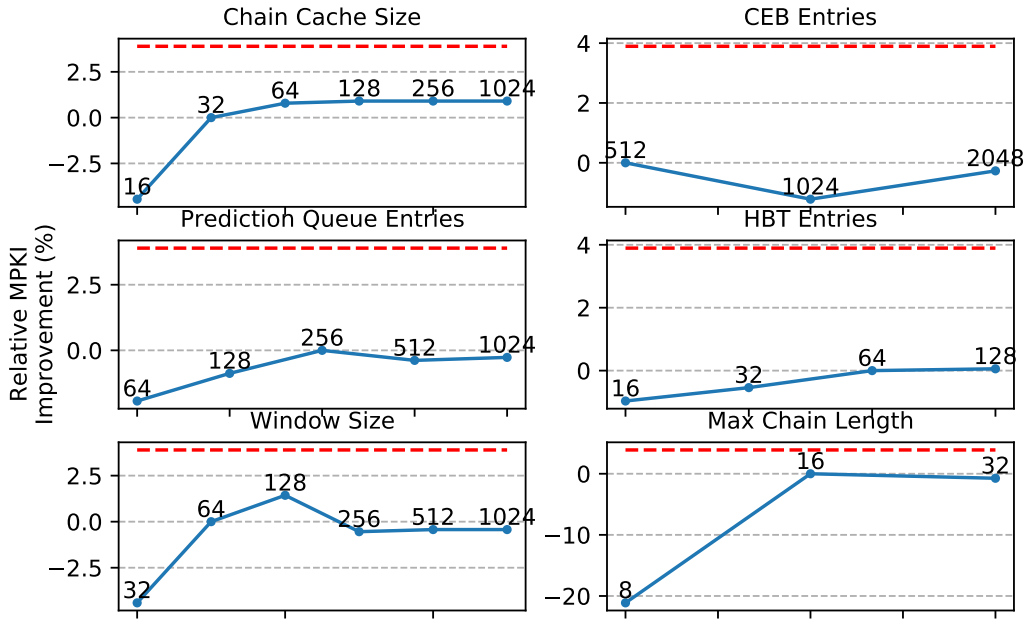

Figure 4.14: MPKI Improvement relative to Mini Branch Runahead. Parameters are swept individually up to the level of Big Branch Runahead (Red dotted line) to show each parameters contribution.

Branch Runahead configuration. On average, Big Branch Runahead improves the MPKI of Mini Branch Runahead by 3.89%. The figure suggests that this improvement is primarily due to the increased window size and chain cache size. Furthermore, the graph suggests that optimal values for window size and chain cache would be 128-entry and 64-entry respectively, meaning that Big Branch Runahead could be implemented using 27KB of total storage.

Energy. Figure 4.15 shows the change in energy due to Branch Ru-

opposed to the 200 Million used for all other experiments.

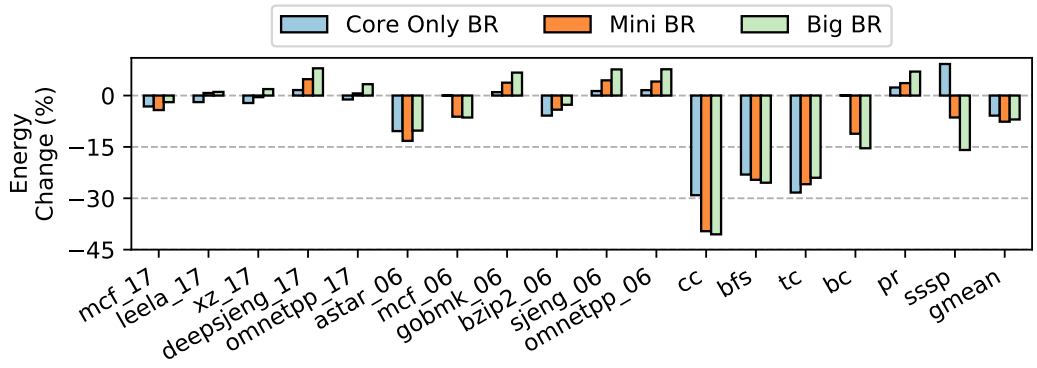

Figure 4.15: Energy Impact (Lower is better)

anhead, as estimated by McPat. Branch Runahead decreases energy on average, primarily due to faster run times. However, Branch Runahead does increase energy usage in two ways. First, there is the increase in static and dynamic power that is generated by new structures. Second, Branch Runahead increases the total number of instructions executed and memory accesses. Figure 4.3 shows the total increase in both ops executed and total memory accesses.

Area. McPat estimates the DCE engine area to be $0.38mm^2$, or about 2.2% of a baseline out-of-order core ($16.96mm^2$ at a 22nm process). Of this total, $0.09mm^2$ is dedicated to the dependence chain cache, $0.15mm^2$ is dedicated to functional units, reservation stations, and physical registers, and $0.14mm^2$ is dedicated to chain extraction and the HBT. [15]

[15]For reference, McPat estimates the 64KB TAGE-SC-L predictor to be $0.73mm^2$. This estimate is a lower bound on the total area, as McPat does not faithfully model the interconnect, muxes, and adders contained within TAGE-SC-L predictor, which consume a

Impact on clock frequency. Branch Runahead minimally affects clock frequency, as almost all units are off the critical-path and are not sensitive to latency. The only component on the critical path is a MUX, which selects between TAGE-SC-L and the DCE engine prediction queues. The DCE executes the dependence chain across many cycles, off the critical path, and inserts the result into a prediction queue. Therefore, processor throughput is minimally impacted.

4.6 Related Work

Gupta et al. [21] target dependence chains that contain one load instruction with a predictable address. While this technique is effective for a subset of branches, Branch Runahead is a more general technique that is able to capture more benefit. Their targeted approach does simplify some hardware (no affector/guards, simplifies chain scheduling), however much of the same hardware is needed to execute the dependence chains.

Farooq el al. [16] propose Store-Load-Branch (SLB) predictor, which predicts data-dependent branches by identifying dependent store-load-branch chains in a program using the compiler. Gao et al. [18] propose a new predictor that targets data-dependent branches by correlating a load's memory address with the result of an upcoming branch. The EXACT predictor [5] also targets data-dependent branches by distinguishing branch instances based on

non-negligible area.

their feeder load's address.

Premillieu et al. [42] save branch results computed on the wrong-path and replay them as predictions later on the correct-path. However, this technique is limited to *control-independent/data-independent* branches that are executed in the shadow of a branch misprediction.

Ayers et al. [7] propose a new methodology to classify the memory access patterns of applications. This technique effectively categorizes dependence chains for load instructions, enabling reasoning about prefetcher timeliness and criticality. Transparent Control Independence (TCI) proposes a method of identifying CIDD instructions that is similar to our method for identifying affector branches [6]. However, their purpose for doing so is completely orthogonal to Branch Runahead. Trace processors was the first work to propose the local/global register concept used by Branch Runahead [48].

Zangeneh et al. propose BranchNet [62] offline training of a CNN to improve prediction accuracy for hard-to-predict branches. However, this technique requires correlation between branch outcomes and history, making the technique less effective for data-dependent branches.

4.7 Future Work

Branch Runahead suffers from two key limitations. The first limitation is timeliness. Despite the initiation strategies presented in this chapter, there is still room for Branch Runahead to execute dependence chains more quickly,

which would further improve MPKI. The second is Branch Runahead's reactive synchronization mechanism. Currently, Branch Runahead waits for a core misprediction to start a dependence chain. In addition, Branch Runahead waits for a second misprediction to recognize that a divergence happens. This essentially means that a dependence chain is guaranteed 2 mispredictions each time it is started. For chains with small lifetimes, these 2 mispredictions can significantly reduce the overall prediction accuracy of the dependence chain.

Trigger points, as used in prior work [10], can be used to eliminate both of these issues. Use of a trigger point (i.e., a static PC in the code that will trigger Branch Runahead to start executing dependence chains) will allow Branch Runahead to start execution without waiting for a branch misprediction. Furthermore, trigger points can be used to start execution at a much earlier point in time, which will in turn help timeliness. The main challenges facing trigger points are 1) determining the correct position of the trigger point and 2) correctly synchronizing data from the backend upon detection of a trigger point.

Lifetime prediction can be used to terminate dependence chains without waiting for a branch misprediction to indicate that the dependence chain has diverged. In some cases, a dependence chains lifetime may be very predictable. For example, in Figure 4.4 it is very clear that the dependence chains in this example will only be accurate for 8 iterations (the for loop terminates after 8 iterations). Therefore, it should be predictable that the lifetime of the dependence chain for Branch A is 8 iterations. Predicting lifetime accurately

will help Branch Runahead terminate dependence chains before encountering the first misprediction. This will of course save a single flush of the backend, but it will also save energy and reduce memory bandwidth as fewer superfluous micro-ops need to be executed by the backend.

Finally, Branch Runahead can be potentially extended to other areas of prediction such as branch target prediction and value prediction. Branch target prediction would be relatively straight-forward for Branch Runahead, as we are already executing the branch instructions themselves. Here, the prediction queues would need to be extended to house the branch targets as well as the predictions. Value prediction is another area where Branch Runahead can potentially provide value. As discussed in section 4.2.2, Branch Runahead differs from traditional Runahead techniques due to its use of affector and guard branches. These enable Branch Runahead to remain accurate while executing continuously. This benefit increases the accuracy of the dependence chains to a level that is appropriate for branch prediction, as opposed to prefetching where simply computing the correct line address is enough. It is likely that value prediction would also benefit from the improved accuracy of Branch Runahead.

Chapter 5

Control Independence

5.1 Introduction

Accurate branch predictors have enabled architects to design wider, deeper pipelines over the past few decades. Unfortunately, advancements in branch predictor accuracy have slowed significantly and are struggling to keep up with the demand for even larger microprocessors [38]. Hard-to-predict branches frequently cause expensive pipeline flushes that not only lower performance, but also waste energy. While improvements in branch predictor accuracy are expected, it is unlikely predictors will improve at a rate that is acceptable for future microprocessors. Therefore, runtime alternatives to branch prediction should be explored.

Control independence is a promising alternative to branch prediction for hard-to-predict branches [12, 13]. With control independence, we predict the merge point of a branch, rather than its direction. Knowledge of the merge point allows us to deploy one of two control-independent strategies: Dynamic Predication or Delayed Fetch. Control-independent strategies do not rely on the branch direction information, and therefore can help maintain high fetch and execution bandwidth and avoid costly flushes. Unfortunately,

each control-independent technique can also cause *performance inversions*, i.e., cases where the performance worsens despite avoiding the branch misprediction. In this chapter, I discuss how to efficiently implement both Dynamic Predication and Delayed Fetch, overcoming critical problems with prior work. Further, I discuss the reciprocal nature of the trade-offs between each of the control independent techniques. I will show that each technique is optimal under different circumstances. As a part of my future work, I will propose a system that can dynamically switch between each technique depending on what is best for the running program.

5.1.1 Dynamic Predication

Dynamic Predication is a runtime mechanism for fetching both paths of a branch, up to the merge point, then executing each path and only committing the results of the correct path. The primary performance goals of Dynamic Predication are to 1) avoid the expensive branch misprediction and 2) do so while minimizing execution latency for the correct path instructions. As the correct path is not known, the optimal strategy is therefore to fetch and execute both paths of the branch.

Auto-Predication of Critical Branches (ACB) [12], the current state-of-the art technique for dynamic predication, is an effective technique for reducing the impact of hard-to-predict branches. Unfortunately, ACB is not effective for all hard-to-predict branches, primarily due to two limiting factors: 1) poor coverage and 2) throttling due to performance inversions. This chapter im-

proves upon ACB in each of these two areas. First, I use a more effective merge point detection algorithm, proposed in Chapter 3, that achieves higher coverage and higher accuracy [44, 45]. Second, I improve upon the ACB microarchitecture by supporting more complex, but important cases, such as nested predication and allowing predicated memory accesses to execute before the branch completes execution.

5.1.2 Delayed Fetch

Delayed Fetch is a runtime mechanism that does not fetch instructions from either side of the branch. Instead, Delayed Fetch predicts the location of the merge point and begins fetching instructions out-of-order from that location. Once the branch has executed and the correct path is known, Delayed Fetch will then fetch the correct path of the branch. Due to the fact that instructions are fetched out of order, Delayed Fetch is also required to rename and allocate[1] instructions out-of-order as well. The primary goal of Delayed Fetch is to avoid the branch misprediction penalty while also maintaining high instruction fetch bandwidth. This is accomplished by fetching post merge point instructions that are highly likely to be on the correct path regardless of the true direction of the branch.

Skipper [13] was the first work to propose fetching instruction out-of-order as an alternative to branch prediction; however, Skipper did not address critical implementation issues, such as the out-of-order branch prediction

[1] Allocate instructions into backend structures; i.e., Re-order Buffer, LD/ST Queue, etc.

	Dynamic Predication	**Delayed Fetch**
Correct-Path Latency	Min.	Max
Fetch Bandwidth	Max	Min.

Table 5.1: Reciprocal trade-offs of control-independent strategies.

problem and complexity issues with out-of-order rename, which has limited its adoption. This dissertation proposes simple and easy to implement solutions to these previously unsolved problems.

5.1.3 The Duality of Dynamic Predication and Delayed Fetch.

Dynamic Predication and Delayed Fetch have reciprocal trade-offs, making each ideal in different scenarios. Table 5.1.2 shows a summary. Dynamic predication minimizes latency of correct path instructions, as both paths are fetched, decoded, renamed, and executed all before the resolution of the branch. However, this low latency comes at the cost of fetch bandwidth and backend resources as *both* paths of the branch must be fetched and allocated. Delayed Fetch, on the other hand, does not fetch either path. Instead, the correct path is fetched after branch resolution when the result of the branch is known. This trade-off minimizes wasted fetch bandwidth and backend resources at the expense of correct path latency.

This dissertation presents an holistic new approach to control independence. I propose a new microarchitecture that has a unified and low cost approach to implementing both techniques. Further, I show that selecting the

correct control independence technique for each application can lead to higher performance than either technique in isolation.

The contributions of this chapter are:

- Solving critical problems related to Dynamic Predication that enable the technique to be used effectively on more hard-to-predict branches. This is accomplished by using the new merge point predictor, as well as supporting features like nested predication.
- Solving critical problems related to the implementation of Delayed Fetch. Previous solutions do not properly update the branch predictor in the face of out-of-order fetch, which lowers performance and makes Delayed Fetch infeasible. In this dissertation we present a new way to predict branches for Delayed Fetch that solves this problem.
- A new unified microarchitecture for implementing both Dynamic Predication and Delayed Fetch. This new microarchitecture allows us to dynamically switch between the two techniques with minimal overhead, giving us the freedom to identify the mode that is best for the running program.
- Identifying the reciprocal nature of the trade-offs between dynamic predication and delayed fetch and demonstrating that both techniques can be used together to create a more holistic replacement for branch prediction for hard-to-predict branches.

5.2 Limitations of Prior Work

5.2.1 Limitations of Compiler Predication

Compilers have been predicating instructions for many years. Predicated execution benefits superscalar processors by 1) eliminating expensive flushes caused by branch mispredictions and 2) eliminating the conditional branch instructions themselves, thereby increasing fetch bandwidth [35]. Unfortunately, predication only improves performance in cases where the branch is hard-to-predict. Identifying hard-to-predict branches can be a difficult challenge for a compiler. The compiler must profile the code, which requires representative input data to be available at compile time. Further, compilers must profile the code on the target micro-architecture, or again the results may differ from what is observed at runtime. Inadvertently predicating an easy-to-predict branch can cause severe performance degradation, which leads compilers to be conservative when doing if-conversion. This ultimately leads to many missed predication opportunities.

Dynamic Predication has the advantage that it can monitor the performance of the actual branch predictor during runtime. This allows the processor to accurately identify the hard-to-predict branches. Further, as discussed in Chapter 2, total branch cost can also be monitored to determine which branches are causing the highest degree of instructions to be flushed and are therefore the best targets for predication.

5.2.2 Dynamic Predication

Diverge-Merge Processor (DMP) [28] uses compiler analysis and profiling to identify candidate branches and their dynamic merge points. This information is appended to the program binary. Then, at runtime, hard to predict branches are identified and dynamically predicated. DMP fetches instructions down both paths of the branch, forking the branch history register and register alias table (RAT). Finally, when the branch resolves and the correct path is known, the correct history register and RAT are identified. Select micro-ops are injected to properly track data-dependences.

Unfortunately, there are several limitations to DMP. First, DMP still relies on the compiler to identify candidate branches, which limits DMP as discussed in section 5.2.1. Second, DMP requires that the branch history register and register alias table be forked to maintain the proper state down both paths of the predication. This creates an unreasonable amount of complexity, and prevents techniques such as nested predication.

Auto-predication of Critical Branches (ACB) [12] identifies hard to predict branches at runtime, then it identified the merge points of hard-to-predict branches using a merge point predictor. Using the dynamically predicted merge points, ACB is able to fetch instructions down both paths of the branch. ACB, however, does not allow both paths of the branch to be executed, instead it waits for the predicated branch to complete execution. At that time, the correct path is allowed to execute and the wrong-path is converted to move instructions that also need to be executed.

ACB addresses many of the shortcoming of prior work by introducing dynamic hard-to-predict branch detection and dynamic merge point prediction. However, the merge point predictor used by ACB memorizes code layout, which in turn lowers coverage, as discussed in Chapter 3. This dissertation uses a new merge point prediction algorithm that significantly boosts coverage, enabling more branches to be predicated. Further, the hard-to-predict branch detection algorithm presented in this dissertation also accounts for branch cost, which detects previously undetected high cost branches that are suitable targets for predication. Further, ACB does not support important cases like nested predication, resulting in lower coverage. Finally, ACB does not allow predicated instructions to execute until after the predicated branch finishes execution. This eliminates one of the key benefits of dynamic predication, which is minimizing latency for correct-path instruction, and reduces memory level parallelism.

5.2.3 Delayed Fetch

Skipper [13] identifies hard-to-predict branches at runtime, then, in cases where the merge point of the branch is the target of the branch, fetches instruction out-of-order to avoid predicting the direction of the branch. Once the direction of branch is known, the skipped path is optionally fetched and inserted into the instruction stream.

Unfortunately, Skipper only targets branches whose merge points are equal to their target address. This eliminates the possibility of using Delayed

Fetch on many branches whose merge point can be trivially predicted. Further, Skipper does not handle cases where there are conditional branches in the skipped region, which result in an out of order branch history register. These complications significantly lower branch predictor accuracy, often negating the positive effect of Delayed Fetch. This dissertation presents methodology for handling branches in the skipped region that does not lower branch predictor performance. Further, my implementation of Delayed Fetch can dynamically switch to fetching predicted paths during full window stalls, which further improves fetch bandwidth. Finally, I use a branch cost table, which identifies all branches that have high cost.

In-order branch prediction with out-of-order Delayed Fetch. Prior work in Delayed Fetch techniques [13] does not address changes to the branch predictor to accommodate out-of-order fetch, which remains an unsolved problem. A typical implementation of Delayed Fetch would present branches to the branch predictor non-deterministically and out of program order, which would in turn lower predictor accuracy. Our implementation of Delayed Fetch solves this problem by predicting branches in the *predicated code order*.

Stark et al. [58] proposes using out-of-order fetch to avoid the latency of i-cache misses. Many of the hardware structures used to implement out-of-order fetch acted as inspiration for this work. However, out-of-order fetch to avoid branch misprediction is a more challenging problem because it is unclear how to modify the branch predictor to handle these cases. Stark was able to

assume the predictor was still predicting accurately, which enabled him to use a decoupled branch predictor to predict addresses after the i-cache miss and begin fetching those addresses. Delayed Fetch, however, cannot simply continue to branch predict as usual, because the existence of a hard-to-predict branch is what is causing the problem in the first place.

5.3 Critical Issues

To date, there remain several critical issues facing Dynamic Predication and Delayed Fetch that limit adoption. Both Dynamic Predication and Delayed Fetch require changes to the branch predictor and instruction fetch stage. Dynamic Predication requires the branch predictor to predict down both paths of the branch. Prior work has suggested that this requires separate history registers for each path [28]; however, as will be shown in this section, that is not the case. Delayed Fetch requires the branch predictor to be able to predict branches out-of-order. Straight-forward implementations of this lead to either information being removed from the branch predictor, or non-deterministic updates to the branch predictor [13]— both of which lead to significant performance loss. This section shows that Delayed Fetch can access the branch predictor in the same way that is used for Dynamic Predication. Unifying the implementation of the branch predictor not only simplifies the implementation of instruction fetch, but also the implementation of rename and the merge point predictor itself.

Rename presents another hurdle for adoption. Prior work requires a

register alias table for each path of the branch [28]. This significant hardware cost prevents techniques like nested predication, which can exponentially increase the number of required register alias tables. Cheaper rename techniques, on the other hand, do not allow instructions to execute until the branch resolves, eliminating one of the primary benefits of predication [12]. This section presents a low cost rename technique that does not sacrifice performance.

This section discusses the implementation requirements for both Dynamic Predication and Delayed Fetch to be successful. The section also introduces solutions to each of the critical issues listed above. Then, section 5.4 presents a unified Control Independent Microarchitecture (CIM) that implements the solutions presented in this section.

5.3.1 Branch Prediction and Instruction Fetch

Predicated Code Order. Dynamic Predication uses the merge point predictor to fetch instructions down both paths of the branch. To achieve this, the processor must view the code *as if* the code were transformed by the compiler. An example of this transformation is shown in Figure 5.1a-b. Each box in this figure (labelled A, B, C, and D) represent different paths, which internally may contain predictable branches. We refer to this new code order as the *predicated order*; i.e., the ordering of the paths after it has been dynamically predicated.

Note, it is not necessary to transform the actual code in memory, we are simply changing the processor's perspective of the original code in memory.

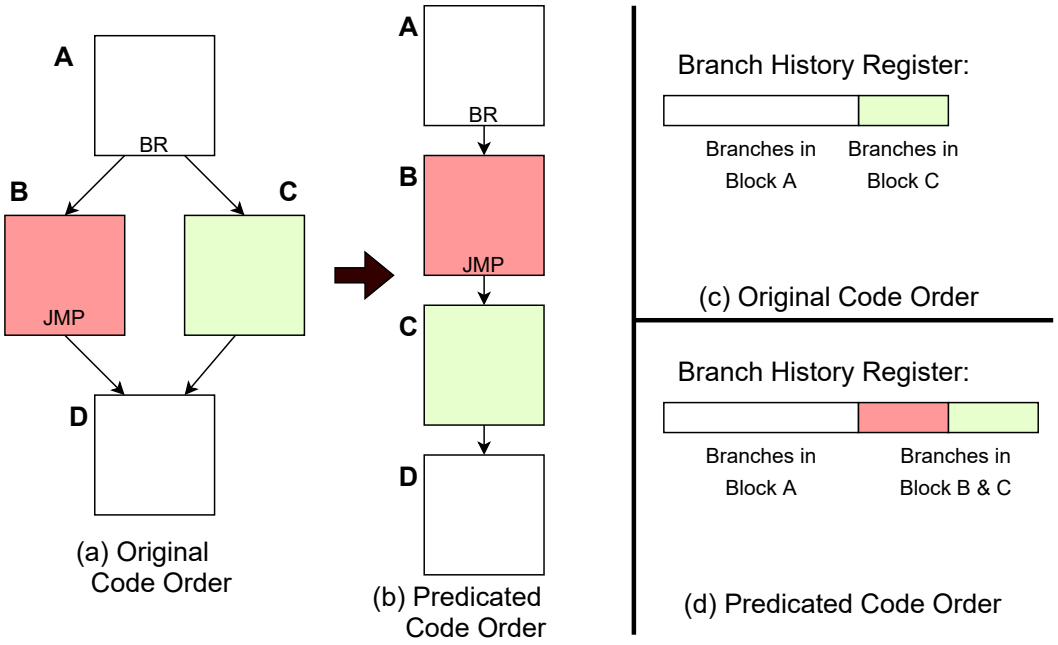

Figure 5.1: Original Code Order (left) and Predicated Code Order (right). Red indicates wrong-path instructions, green is correct-path.

This is achieved by installing an entry for the branch in the merge point predictor. Upon a merge point predictor hit, the merge point predictor will supply both the target and fall-through addresses of the branch, as well as a predicated order bit indicating the ordering of the paths.

The order of the paths is determined by the location of the *jumper branch*; i.e., the branch that jumps to the location of the merge point (contained at the end of Path B in Figure 5.1a-b). This order is selected such that the path that *does not* contain the jumper will continue to fall-through to the merge point. This optimization helps to not introduce new packet breaks during instruction fetch/branch prediction.

Re-training the Branch Predictor. Once an entry for the branch has been installed in the merge point predictor, the frontend of the pipeline will begin processing the code according to the predicated order. This means that the branch predictor will observe branches in the predicated code order rather than in the original code order. The new ordering will be unfamiliar to the branch predictor and will require a new warm up period.[2]

In addition to the warm up period, the predicated order introduces a new variable into the history register: wrong path branch instructions. As shown in Figure 5.1c-d, the predicated code order causes wrong path branch instructions to be inserted into the history register and, unlike with the original order, these wrong-path branches will not be flushed out or recovered later. While it is possible to track the wrong-path branches and eventually repair the history, doing so would be expensive and unnecessary. In theory, the addition of wrong-path branches can either create new sources of noise or new sources of correlation, depending on the code. At worst, the wrong-path branches amount to more noise in an already very noisy history register. The branch predictor is already capable of dealing with such sources of noise. In practice, we observed negligible impact on branch prediction accuracy from leaving the wrong-path branches in the history.

Fetch Order. The predicated code order is determined based on the location of the jumper branch. Choosing this order minimizes packet breaks,

[2]Note, due to the warm up period caused by the code transformation, we do not want to switch between views of the code often.

which in turn improves branch predictor throughput. However, the predicated paths themselves have no order relative to one another and can therefore be fetched/renamed in any order. This allows us to predict which path should be fetched first in order to minimize the latency of correct path instructions. The correctness of the prediction does not matter, as either fetch order is valid. However, when the correct path is correctly predicted, fetching and renaming that path first leads to a small performance improvement.

The Out-of-Order Branch Prediction Problem Implementations of Delayed Fetch result in severe drops in branch predictor accuracy. Even previously easy-to-predict branches can begin to suffer from high misprediction rates. This drop fundamentally occurs due to the fact that the branch predictor is now expected to predict branches out-of-order, something it was never designed to do. I attribute three reasons for the decline in branch predictor accuracy:

1. No prediction for the Delay Fetch branch. No prediction is made for the Delay Fetch branch, and therefore there is nothing to shift into the branch history register. This causes information to be removed from the history register, which may adversely effect some branches. Note, Dynamic Predication suffers from the same problem. This problem is fundamental to strategies that seek to avoid predicting hard-to-predict branches.

2. No predictions for any branches down either path of the branch. Delayed Fetch skips over both paths of the branch in order to conserve fetch bandwidth. This unfortunately means that we are also skipping over all of the

branches in these code blocks, which means that the branch history register is losing even more information. For example, when control reaches Path D (Figure 5.1), the branch predictor will be missing history information on all the branches in Path B or C. If any of the branches in Path B or C are correlated to the outcome of branches in Path D, then accuracy will suffer.

3. Non-deterministic execution latency of the Delay Fetch Branch. Eventually, the Delay Fetch branch finishes execution and we can begin fetching down the correct path of the branch (Path C in Figure 5.1). This creates two challenges: 1) We will need to make predictions for all branches in Path C. This is problematic because the history register currently contains some of the branches from Path D. The exact number of Path D branches depends on the execution latency of the Delay Fetch branch. For example, if the Delay Fetch branch executes quickly, then only a small number of the branches in Path D will be shifted into the history register. However, if the Delay Fetch branch takes a longer time to execute, then more branches will be shifted into the history register. This means that the exact contents of the history register depend on the execution latency of the Delay Fetch branch, which is non-deterministic. This creates many possible histories that the branches in Path C can observe, which increases the storage requirements and warm up time for TAGE to memorize all history-outcome pairs. 2) Once we have fetched Path C, we will need to decide if we want to add the branches in Path C to the branch history. Both options are problematic. If we add the branches to the history we will be inserting them into the history at a non-deterministic point

in time, which means that branches in Path D will sometimes observe the branches in Path C and sometimes not. However, if we do not insert the Path C branches, then we will forever lose potentially valuable history information.

Prior work addresses these problems by limiting Delayed Fetch to only target branches that contain biased branches in their skipped regions. Bias branches do not need to access the branch predictor, which solves the out-of-order branch predictor problem. Unfortunately, this solution significantly limits the number of branches that can use Delayed Fetch, making the technique far less useful.

Delayed Fetch - Predicated Code Order.

I propose predicting branches according to the predicated code order for Delayed Fetch. Doing so solves two critical issues associated with out-of-order branch prediction. First, predictions are generated for both Paths B and C. This means that the history register is no longer missing potentially critical information from those blocks. Second, all branches are inserted into the history register at deterministic points in time (i.e., when the branch is predicted rather than after execute), which creates repeatable behavior and allows the branch predictor to learn more efficiently.

Generating the predictions for both paths has the added benefit of letting the branch predictor tell us which paths are in the skipped region on the Delay Fetch branch. As will be discussed later in the section, this allows us to more accurately compute the register live-in/out information for the skipped

region, which in turn simplifies rename. Furthermore, this information can be used to compute the number of dynamic instructions between the Delay Fetch branch and the merge point, which can in turn be used to verify the correctness of the merge point or detect a divergence. This means the merge point predictor no longer needs to predict the merge point distance or the register live-in/out information, which simplifies the design of the merge point predictor.

Delayed Fetch - Fetch Order. For both Dynamic Predication and Delayed Fetch, the branch predictor will traverse both paths of the branch. However, the difference between these two techniques is the fetch order. While Dynamic Predication will fetch instructions down both paths of the branch, Delayed Fetch skips the fetch of these instructions and instead fetches instructions from beyond the merge point. This is accomplished by simply ignoring the predicted addresses for the skipped region of code. Rather than passing the skipped region addresses to the fetch stage, these addresses are cached in a new structure called *the prediction cache* (discussed more in section 5.4. Addresses from beyond the merge point are then passed to the fetch stage, which will begin fetching instructions from beyond the merge point. Eventually when the Delay Fetch branch resolves, the true direction of the branch will be broadcast to the prediction cache. The prediction cache, which cached the previously generated branch prediction for both paths of the Delay Fetch branch, will then pass the prediction for the correct path to the fetch stage, which will then fetch the correct path instructions.

Importance of decoupled branch prediction. Unifying the implementation of the branch predictor simplifies many of the complexities associated with Delayed Fetch. Unfortunately, generating branch predictions down both paths wastes branch predictor bandwidth. This goes against the goals of Delayed Fetch, which purports to preserve fetch bandwidth. To mitigate this, we require the implementation of a decoupled branch predictor [46], which typically generates predictions at a higher rate than the fetch rate. Therefore, the wasted prediction bandwidth is usually hidden.

While the decoupled branch predictor design does mitigate the effects of predicting down both paths, it does not completely eliminate it. Ideally, we would not waste branch predictor bandwidth predicting down both paths for Delayed Fetch. However, generating these predictions simplifies critical issues with out-of-order branch prediction, instruction fetch, and rename, which ultimately make the trade-off worth it.

5.3.2 Rename

Both Dynamic Predication and Delayed Fetch present challenges for a traditional rename implementation. In this chapter, I present a unified rename implementation that works for both Dynamic Predication and Delayed Fetch. Having a single unified implementation is ideal because it minimizes the hardware cost and complexity— rename does not need to know whether the code is using Dynamic Predication or Delayed Fetch. Furthermore, rename must also scale for nested Dynamic Predication and Delayed Fetch. Our experi-

ments show that supporting a nested depth of 5 branches leads to significant improvement in some workloads. Prior work would require 32 register alias tables to support a nesting depth of 5. Our design seeks to minimize this cost.

Fortunately, the unified design of the branch predictors helps us accomplish both of these goals. For both Dynamic Predication and Delayed Fetch, the branch predictor produces predictions down both paths of the branch. This in turn gives us the complete set of basic block addresses that will be fetched down both paths. Using a new structure called *the map cache*, we can use these basic block addresses to look up register live-in/outs for each basic block. The basic block live-in/out information allows us to trivially compute the live-in/out information for each path of the branch. Using this information, the rename unit can rename the live-in/outs of the predicated or delayed fetch region of the branch *prior* to renaming the instructions within the region. This achieves two things. First, it unifies the rename algorithm. Both Dynamic Predication and Delayed Fetch will first rename the live-in/outs of region, then at a later point rename the instructions within the region itself. Second, it minimizes the hardware cost. Because the live-in/out register of the region have already been renamed, the register alias table itself can be used as a temporary scratch space that can assist in the rename of each path. The hardware required to do this as well as a more detailed description of the rename algorithm will be presented in the next section.

5.3.3 Gap Allocation, Deadlock, and Full Window Stalls

Delayed Fetch skips over the predicated code block, instead fetching post merge point instructions that are more likely to be on the correct path. Unfortunately, this also implies that instructions must be allocated into the Re-order Buffer (ROB) out-of-order as well. This requires a gap region to be allocated in the ROB for each Delay Fetch branch. The gap reserves space for the skipped instructions, which prevents deadlock from occurring. Unfortunately, allocating gaps in the ROB can cause the ROB to fill up more quickly, which can increase full window stalls. During a full window stall, the frontend is forced to stop fetching altogether, in turn lowering effectice fetch bandwidth. During this time, as no useful instructions can be fetch, Delay Fetch can begin predicting the result of the Delay Fetch branches that have already allocated a gap in the ROB. The full window stall does not apply to instructions in these regions, because space for them in the out-of-order window has already be reserved. Therefore, during a full window stall, branches in the ROB will trigger the fetch of their predicted path. In the event the prediction is correct, then otherwise wasted fetch bandwidth is used to fetch the correct path of the branch early. If the branch is mispredicted, then the ernoiously fetched instructions can be flushed, and the correct path can be fetched/filled at that time. It is worth noting that mispredictions during a full window stall do not waste additional fetch bandwidth, because the full window stall was already preventing the fetch of useful instructions.

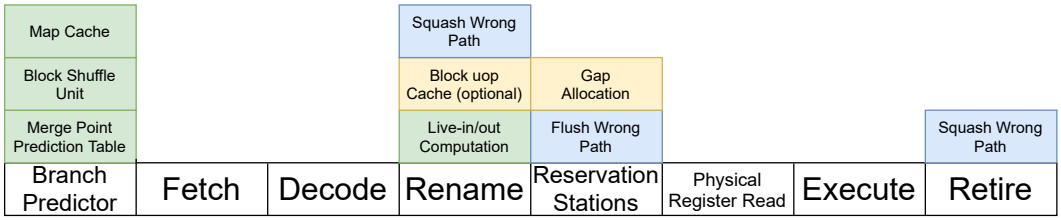

Figure 5.2: Changes to the pipeline. Blue indicates logic required for Dynamic Predication, yellow indicates logic needed for Delayed Fetch, and green indicates logic needed for both.

5.4 Control Independent Microarchitecture

The reciprocal nature of the trade-offs between Dynamic Predication and Delayed Fetch motivate the design of a unified control independent microarchitecture that is capable of switching between the two strategies with minimal overhead. Both strategies require changes to the branch predictor, fetch, rename, allocate, and retirement stages of the pipeline. Figure 5.2 summarizes these changes, with blue indicating changes needed for Dynamic Predication, yellow indicated changes needed for Delayed Fetch, and green indicating changes needed for both.

5.4.1 Merge Point Prediction

The merge point predictor consists of five components: a Hard Branch Table (HBT) that is responsible for detecting hard-to-predict branches (presented in Chapter 2), a Wrong Path Buffer (WPB) that is responsible for detecting merge points (presented in Chapter 3), a Prediction Table that is responsible for supplying the correct prediction in the instruction fetch phase

Structure	# Entries	Per-entry Composition
HBT	64-entry	partial-tag (12b), Cost Counter (8b)
WPB	128-entry	partial-tag (12b)
Prediction Table	16-entry	Valid (1b), Partial-tag (12b), Merge Address (48b), Confidence (5b), Bias Pred. (2b), Predicated Order (1b), Mode (2b), Involve counter (4b)
Training Table	1-entry	Valid (1b), Merge address (48b), Instruction counter (6b)
Active Context	5-entry (shared)	Path (1b), Predicated Order (1b), Br Fall-through (48b), Br Target (48b), Merge address (48), Instruction counter (6b)

Table 5.2: Merge Point Predictor Structures

(presented in Chapter 3), a Training Table that is responsible for warming up new predictor entries, and an Active Context Stack that is responsible for monitoring active merge point predictions. The Training Table and Active Context Stack replace the Update List presented in 3. The Training Table requires that a merge point demonstrate a specified accuracy before being used for prediction, which decreases the number of divergences, and the Active Context Stack allows for nested merge point predictions. Figure 5.3 shows a block diagram of each of these components together and Table 5.2 summarizes each components size. This section describes each component of the merge point predictor. In some cases, components are sized differently than they were in Chapter 3. This is primarily due to tuning differences between the different set of workloads used in each chapter. The reason behind any other differences will be described in the sections below.

Hard Branch Table (HBT). Hard-to-predict branches are detected using the HBT, which was presented in Chapter 2. In this chapter, we use the

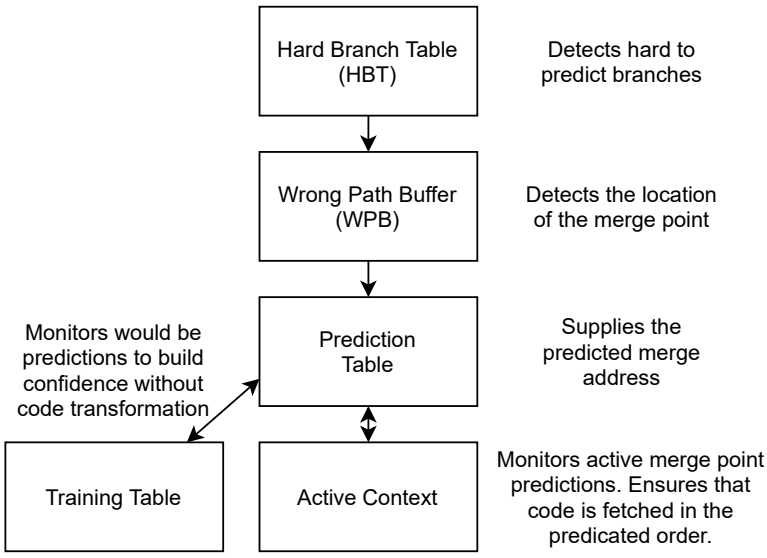

Figure 5.3: Merge Point Predictor.

version of the HBT that tracks total branch cost. Upon a branch misprediction, the number of instructions that are flushed from the ROB are counted.[3] When the mispredicting branch is retired, that total is added to the counter of the entry corresponding to the mispredicting branch in the HBT. The parameters used for the HBT are a Period of 10,000, an Acceptable Cost Rate of 5%, and Probability of a False Positive of 1%. Using the calculations presented in Chapter 2, this leads to a periodic decrement rate of 500 every 10,000 mispredictions, and an 8-bit cost counter.

Wrong Path Buffer (WPB) and Balanced Stack Counter. The WPB works by intersecting the wrong-path and correct-path of a branch dur-

[3]This can be done easily by subtracting the ROB address of the branch from the tail pointer.

ing a branch misprediction. The wrong-path is inserted into the WPB when a branch that hits in the HBT triggers a branch misprediction. Moving forward, each retired correct-path instruction looks up the WPB. A hit indicates an intersection between the correct-path and the wrong-path, which is predicted to be a merge point. In this paper, we use the WPB design as it is presented with two modifications.

First, we add a *balanced-stack counter*. The balanced-stack counter is used to detect false merge points that can appear due to call and return instructions. For example, if both sides of the branch call the same function or return to the same point. These cases present challenges for our control independence strategies, and should be omitted from merge point detection. To detect these cases, we add a signed counter to the WPB. The counter is incremented on call instructions and decremented on return instructions. We only allow merge points to be detected when the counter is 0.

Second, we task the WPB with locating the *jumper* branch. Recall that the predicated order of the code is determined by the location of the jumper. The WPB is also responsible for finding the jumper branch and determining the predicated order.

Prediction Table. Once the WPB detects a merge point, that address is saved in the prediction table. The contents of each prediction table entry are detailed in Table 5.2. The prediction table is indexed by the branch address during the branch predictor stage of the pipeline. If there is a hit and the matching entry has not been throttled, that indicates that we have a valid

merge point prediction for this branch and will deploy one of the control independent strategies for this branch (specified by the mode bits). If the entry is confident (confidence counter is saturated), then we will activate the specified control independent strategy by inserting the entry into the Active Context. If the entry is not confident, then we will train the entry further by inserting it into the Training Table.

Training Table. The training table is a single entry table intended to build confidence without activating a control independence strategy. When an entry is inserted into the training table, addresses coming out of the branch predictor are scanned for the merge point. If the merge point is found within the learning interval, the confidence counter is incremented. If the merge point is not found within the learning interval, then the confidence counter is decremented by 15.

Active Context. The Active Context ensures that the branch prediction stage traverses the code using the predicated order; i.e., once the merge point is reached on the first path, the branch predictor is redirected down the second path of the branch until the merge point is found for a second time. The Active Context is organized as a 5-entry stack. each time a branch hits in the merge point predictor a new entry is pushed onto the active context. In the event of nested predication, the Active Context stack depth indicates the maximum nesting depth. Any nesting branches beyond that depth cannot be merge point predicted. The Active Context traverses down each path in a depth first manner.

Only a single merge point address is maintained for the entire Active Context stack, meaning that all nested merge point prediction inherit the same merge address prediction from the parent. While this limitation is not necessary, it reduces the overall complexity of nesting.

Divergences. When the predicted merge point address is not found within the learning interval, a divergence is triggered. Divergences are detected as soon as the branch predictor passes the learning interval without finding the merge address. Note that this may occur before the execution of the predicated branch. Once a divergence is detected, the front-end is immediately halted (to save energy), however control is not directed down the correct path until the predicated branch finishes execution.

5.4.2 Changes to the Branch Predictor Stage

Figure 5.4 summarizes the changes to the Branch Predictor pipeline stage. New structures are highlighted in green if they are needed for all control independent strategies and yellow for structures only needed for Delay Fetch.

Shuffle Unit and Prediction Cache. The *shuffle unit* is responsible for filtering and reordering predictions from the branch predictor. The branch predictor must observe the same branch order so that it can properly train. For this reason, the branch predictor will always observe the same predicated order for all control independence strategies. The shuffle unit then re-orders predictions (in the case of dynamic predication) or filters predictions (in the case of Delayed Fetch) as appropriate for the given mode and branch bias

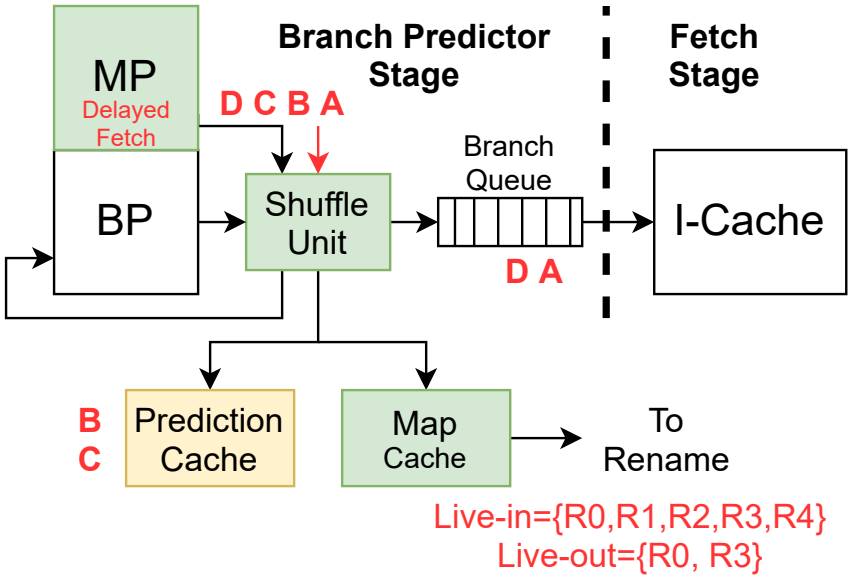

Figure 5.4: Changes to the Branch Predictor Stage. The red text is an example highlighting the difference between branch prediction order and fetch order.

prediction supplied by the merge point predictor. Figure 5.4 shows an example (red). Here, the branch predictor encounters the code shown in Figure 5.1. The branch predictor and merge point predictor traverse the code in the predicated order: A, B, C, D. These paths are passed to the shuffle unit in that order. The merge point predictor specifies that Paths B and C are to be Delay Fetched, which causes the shuffle unit to filter-out Paths B and C, saving their branch predictions into the *prediction cache*, while passing along the branch prediction for Path A and D to the fetch stage. Later, when the branch in Path A finishes execute, the true direction of the branch is broadcast to the shuffle unit, which then supplies the predictions generated for the correct path (either B or C).

Map Cache. The *map cache* saves live-in/out information for each

basic block in the predicated region of a branch (i.e., for each basic block that makes up Path B and C). Live-in/out information is required for fast-path rename (discussed below) during Dynamic Predication and required in all cases for Delayed Fetch. A map cache hit only occurs if all of the basic blocks in the predicated region hit in the map cache. If even one block misses, then the whole region has missed in the map cache, which causes slow-path rename in the case of Dynamic Predication or a divergence in the case of Delayed Fetch.

If all basic blocks in the predicated region hit in the map cache, then the live-in/out vectors for each basic block are accumulated into a single live-in/out vector for the entire predicated region. Figure 5.4 shows an example. Note that all live-outs are also added to the live-in vector. This is due to the fact that we will eventually issue CMOVs for each live-out, thereby sourcing each register in the live-out vector.

The Map Cache allows the microarchitecture to compute the live-in/out's of the predicated region early, before enterring the rename stage. This allows the rename stage to only worry about renaming the live-in/outs correctly. Once this has been done, the two predicated paths can be renamed in any order or at a later point in time, which makes the rename unit impervious to the code re-ordering that occurs in the branch predictor stage.

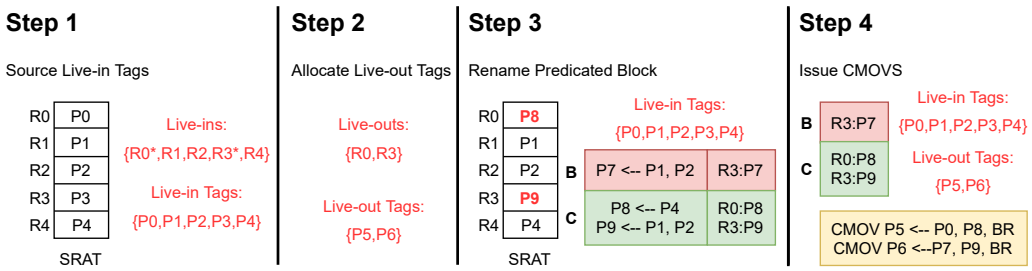

Figure 5.5: Fast Path Rename.

5.4.3 Changes to Rename

Renaming predicated instructions either takes a slow-path or a fast-path, depending on whether the predicated region hit or missed in the map cache.

Slow-path Rename. Slow-path Rename (valid only for Dynamic Predication) involves minimal new hardware, but may take a longer latency depending on the pre-existing checkpoint/recovery mechanism in the baseline microarchitecture for the Speculative Register Alias Table (SRAT). The steps for Slow-path Rename begin when the predicated paths enter the rename stage. In Figure 5.1, this would happen when Path B or C reach the rename stage.

Rename for Path B is performed in the usual way. Additionally, new hardware computes the live-in/out's for every basic block in the path. Once a basic block passes through rename completely, the live-in/outs of that basic block are saved into the map cache. After Path B finishes rename, a recovery of the SRAT is triggered. Depending on the pre-existing hardware for recov-

ering the SRAT, this may take 1 or more cycles (hence the name "slow-path rename"). After this recovery is complete, the process of renaming Path C may begin. Again, this process is the same as baseline rename, but we are again computing/saving the live-in/outs for the basic blocks in Path C.

Issuing CMOV uops. Once Patch C has been renamed and Path D enters the rename stage, we will see that the predicted merge point was found and we will use the live-out registers computed for each path of the branch to inject CMOV uops. The CMOV uops take 3 operands: the physical register tag from the not-taken path, the tag from the taken-path, and tag generated for the predicated branch. When the predicated branch finishes execution, it will notify the CMOV uop in the reservation station, and the correct-path tag will be selected (if ready) for the move.

Fast-path Rename and the Live-In/Out Tag Cache. Fast-path Rename occurs when all of the basic blocks in Path B and C hit in the map cache. [4] Once the live-in/out information has been accessed for each basic block, the accumulated live-in/out masks for Path B and C can trivially be computed by taking the union. The accumulated live-in/out information will allow for a faster rename process, which is summarized in Figure 5.5. Note, step 3 is simply the typical rename step that Paths B and C will go through anyways. The only new latencies are 1) the latency to inject CMOVs (step 4), which is necessary for predication, and 2) the latency to source the Live-in

[4]Note that the map cache is accessed in the branch prediction stage. See Figure **??**.

Tags (step 1), which will consume SRAT bandwidth and must be completed before step 3 can begin.

Another advantage of fast-path rename, is that it does not require steps 3 and 4 to be done immediately. Thus, fast-path rename also gives the ability to implement out-of-order rename for Delayed Fetch and Delayed Predication. When a Delay Fetch branch reaches rename, the accumulated live-in/out mask (computed during the branch prediction stage, when the basic blocks in each path identified) can be used to complete steps 1 and 2. The results of these steps (live-in/out tags) are stored in the live-in/out tag cache, which will hold the tags until the correct-path of the Delay Fetch branch reaches rename.

5.4.4 Squashing wrong-path instructions

Dynamic predication requires that wrong-path instructions be squashed once the correct-path of the branch is known. This happens lazily, either when wrong-path instructions are retired or reach the rename stage (if the predicated branch finishes execution before the predicated block reaches rename). Wrong-path instructions that are currently executing or waiting in the reservation stations are immediately flushed.

5.4.5 Out-of-order Rename

Out-of-order Rename works similarly to Fast-Path Rename in Dynamic Predication. A map cache hit is required to enable Out-of-order Rename. If any of the basic blocks miss in the map cache, then the Delayed Fetch is

aborted, as if a merge point divergence was detected. If all basic blocks hit in the map cache, then that information is passed along to rename. The live-in/outs are used to save the tags for the live-in register and allocate new tags for the live-out register of the gap region (i.e., skipped micro-ops). Once new tags have been generated for all live-outs, the post merge point instructions can be renamed in the usual way.

Eventually, when the Delayed Fetch branch resolves, the skipped instructions will pass through rename. At this point the saved live-in tags can be used to rename the correct-path gap micro-ops. Finally once all instructions in the gap have been renamed, move micro-ops are injected into the instruction stream to move the live out tags of the recently renamed gap micro-ops to the live-out tags that were generated before the rename of the post merge point instructions. The generation of these move instructions effectively patches the "hole" in the data-flow graph.

5.4.6 Gap Allocation and Tracking

Delayed Fetch and Delayed Predication require that a gap is allocated in the ROB, reservation stations, load/store queue, and any other structures that require in-order allocation. This gap reserves space for the delayed instructions which do not have the ability to allocate in-order, thus preventing deadlock. Contrary to prior work [42, 13] that allocates enough space for the larger of the two paths of the branch, our implementation allocates a fixed size region. This region is the managed by the Gap Tracking Unit, which tracks

all allocated gaps in the backend, and the current number of micro-ops that occupy them. Allocating a fixed size gap gives us more control: allocate a gap too big and we waste precious backend resources. Allocate a gap too small and we unnecessarily limit ILP.

In the event of a full window stall, any unfilled gaps are identified, and a prediction for the corresponding branch is sent to the prediction cache. This triggers the fetch of the Delayed Fetch instructions early, which utilizes otherwise unused fetch bandwidth. If the prediction is later found to be incorrect, the incorrect Delay Fetch instructions can be flushed, and the correct path can be fetched at that time.

5.4.7 Waste-based Throttling

Dynamic Predication and Delayed Fetch are best when the average waste due to either mechanism is less than the average waste due to branch mispredictions. Chapter 2 defines the waste due to branch mispredictions to be the number of micro-ops flushed from the ROB during a branch misprediction. Tracking this total over time indicates how much a branch impacts performance based on the frequency and latency of mispredictions. Dynamic Predication can lead to significant waste due to the fact that Dynamic Predication causes both paths to be fetched each time the branch is encountered. Furthermore, CMOV micro-ops must be generated to stitch together the dataflow graph. Each of these factors can contribute significant waste in extreme circumstances. Therefore, a field is added to each merge point predictor entry

Core	16-Wide Issue, 1024-Entry ROB, 256-Entry Reservation Station, 3.2 GHz, 64KB TAGE-SC-L Branch Predictor [52]. Modeled by Scarab [2].
WPB	128-entry, 4-way, max merge point distance 40 uops.
L1 Caches	32 KB I-Cache, 32 KB D-Cache, 64 Byte Lines, 2 Ports, 3-Cycle Hit Latency, 8-Way, Write-Back.
L2 Cache	2 MB 12-Way, 18-Cycle Latency, Write-Back.
Memory Controller	64-Entry Memory Queue.
Prefetchers	Stream: 64 Streams, Distance 16. Prefetch into LLC.
DRAM	DDR4, 8Gb, x8, 2400R, Modeled by Ramulator [30].

Table 5.3: Baseline Configuration

that tracks waste due to Dynamic Predication. Waste is tracked using a single per-branch saturating counter that counts squashed wrong-path and extra CMOV micro-ops. The waste counter is periodically decremented similar to the branch misprediction waste counter in the Hard Branch Table (HBT). When a branch hits in the merge point predictor table, the Dynamic Predication waste counter is compared to the waste counter from the HBT. This effectively compares the waste generated by Dynamic Predication due to the waste generated by branch mispredictions. Whichever counter is lower corresponds to the technique that likely generates less waste overall. If less waste is generated by branch mispredictions, then the merge point prediction is throttled and the branch is not predicated. While waste-based throttling primarily benefits Dynamic Predication, whose primary drawback is wasting fetch/execution bandwidth, waste-base throttling also provides minor improvement to Delayed Fetch. In that mode only the extra MOVE micro-ops are considered waste.

5.5 Results

5.5.1 Evaluation Methodology

To simulate our proposal, we use Scarab [2]— an open source simulator commissioned by Intel in their Intel/NSF FoMR initiative [1]. Scarab is an execution-driven, cycle-accurate x86 simulator whose front-end is based on PIN [34]. The simulator faithfully models core microarchitectural details, the cache hierarchy, wrong-path execution, and includes a detailed non-uniform access latency DDR4 memory system, modeled by Ramulator [30]. We model the 64KB TAGE-SC-L [52] branch predictor with the configuration submitted to CBP-2016. The 64KB TAGE-SC-L is the best known realistic branch predictor. The branch predictor is modeled as a decoupled branch predictor which sits in its own stage prior to the fetch stage. The branch predictor populates a prediction queue that sits between the branch prediction stage and the fetch stage. The queue is capable of holding 64 predictions. The branch predictor faithfully models all packet break situations. Our baseline models a 16-wide machine (decoder outputs up to 16 micro-ops) with an instruction window of up to 1024 micro-ops. We model an aggressive baseline to emphasize the use cases where we see Dynamic Predication and Delayed Fetch being most valuable. Table 5.3 describes our system.

Dynamic Predication Configuration. To simulate Dynamic Predication, we use a merge point predictor with a predictor context stack depth of 5 (i.e., supports nested predication of depth 5) and a learning interval of 40 micro-ops.

Delayed Fetch Configuration. To simulate Delayed Fetch, we use the same configuration as Dynamic Predication with a maximum gap size of 20 micro-ops for allocating gaps into the backend.

Benchmarks. We evaluate Dynamic Predication and Delayed Fetch on SPEC CPU2017 Integer Speed, SPEC CPU2006 Integer [3] and GAP Benchmark Suites [8]. From that set, we select the branch misprediction intensive benchmarks with an average MPKI greater than 2. We use the SimPoints [41] methodology to identify anywhere between one to five representative regions per benchmark. We run each region for 200 million instructions, then compute the weighted average of all the regions. We run SPEC benchmarks on the ref input set, and use *-g 19 -n 300* inputs for GAP. If there is more than one ref input, then the benchmark is run on each input, and a weighted average, weighing each input by the total dynamic instruction count, is used to compute a single metric for the entire benchmark.

Energy. We model chip energy using McPAT [33]. In addition to McPat, we also supply the reduction in off path micro-ops executed to further demonstrate the expected reduction in energy.

Metrics. We use Instructions Per Cycle (IPC) as the performance metric and Branch Mispredictions Per Kilo Instruction (MPKI) to evaluate improvements in prediction accuracy. Coverage is computed by looking at the reduction in branch predictor MPKI due to accurate merge point predictions.

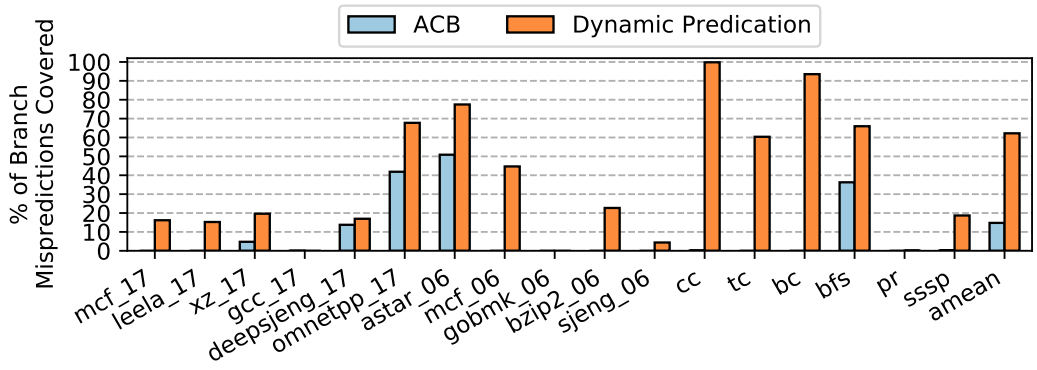

Figure 5.6: Percentage of Branch Misprediction covered by the ACB vs our merge point predictor

5.5.2 Coverage Results

The first step to Dynamic Predication and Delayed Fetch is to identify hard-to-predict branches and accurately detect their merge point. Figure 5.6 shows the percentage of branch mispredictions covered by ACB [12] and the merge point predictor presented in this chapter. As the figure shows, our implementation covers an average of 62.2% of branch mispredictions with an accurate merge point prediction, while prior work only covers 14.8%. As discussed in Chapter 3, this is primarily due to the code layout memorization approach used by prior work to detect merge points. Our approach compares the correct path and the wrong path during a branch misprediction recovery to find the intersection. This is a much more generic way of detecting merge points that results in higher coverage. Furthermore, our merge point predictor makes use of an Active Context Stack, which allows for nested merge point predictions. This further improves the coverage numbers as the merge point

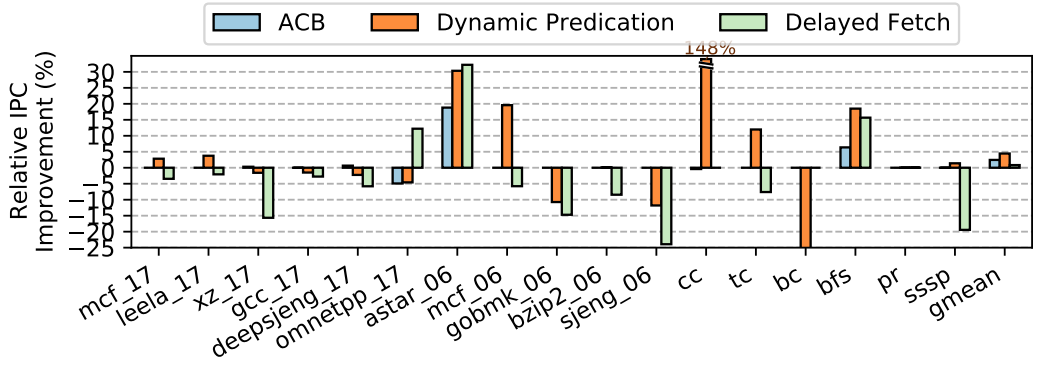

Figure 5.7: IPC Improvement of ACB, Dynamic Predication, and Delayed Fetch without throttling techniques.

predictor is able to accurately predict in these nested cases. ACB, on the other hand, does not support nested predication, which limits their coverage in these cases.

5.5.3 Performance Results

Figure 5.7 shows a summary of the IPC improvement for ACB, the prior dynamic predication technique, our implementation of Dynamic Predication, and our implementation of Delayed Fetch. For purposes of showing results in this dissertation, throttling mechanisms in both the prior work and our work have been disabled. This will allow us to discuss both the positive and negative results in a clearer way, which will allow us to discuss future work required to make these techniques more effective.

Dynamic Predication. As shown in Figure 5.7, Dynamic Predication (without throttling) results in an average IPC improvement of 4.5%,

compared to ACB which improves IPC by 2.5%. Dynamic Predication largely improves over ACB for two reasons. First, the improved coverage leads to more opportunities to use dynamic predication, which in turn leads to fewer branch mispredictions in those workloads. Second, a positive prefetching effect is caused by Dynamic Predication. ACB does not benefit from a similar prefetching effect due to the fact that they do not allow predicated instructions to execute.

The negative outlier workloads generally occur for two reasons. First, the waste generated by the predication (i.e., wrong-path predicated instructions, CMOV instructions) exceeds the average waste generated by branch mispredictions (i.e., wrong-path instructions). This can happen when a branch with a low misprediction rate and/or a relatively low execution latency is predicated. This is what is causing both the *gobmk_06* and *sjeng_06* workloads to suffer. The waste-based throttling mechanism targets cases like this; however, as these workloads show that throttling mechanism is not perfect.

Second, a poorly predicted merge point address, as is the case in *bc*, can result in divergences, which are essentially the same cost as a branch misprediction. Generally, these cases are aggressively throttled, which makes them largely not a problem. However, in some cases, like *bc*, the predicted merge-point is actually correct and does not lead to a divergence, but rather the predicted merge point is not optimal and leads to poorly predicated code. This most commonly occurs for loop branches, where the merge point predictor usually does not identify the optimal location of the merge point. Again, the

future work section of this chapter will address this issue in more detail.

Delayed Fetch. Figure 5.7 shows that Delayed Fetch (without throttling) results in an average performance improvement of 0.9%. Most notably, Delayed Fetch improves performance over Dynamic Predication in two workloads: *omnetpp_17* and *astar_06*. While this result demonstrates the hypothesis of this chapter, Delayed Fetch overall results in more negative outliers than Dynamic Predication. While some of this would be mitigated by dynamically switching between Dynamic Predication and Delayed Fetch, it is worth discussing the negative Delayed Fetch results on their own so that future work in Delayed Fetch can be discussed.

Omnetpp_17 contains three nested hard-to-predict branches that make up a large fraction of the overall mispredictions in the workload. These hard-to-predict branches also guard critical memory accesses that are not correctly pre-fetched by the wrong-path of the branch or hardware prefetchers. Therefore, improving the speed at which the correct path is fetched, decoded, and executed improves the overall critical path of the benchmark, which in turn results in critical memory accessed getting to the memory system sooner. *Astar_06* contains many nested hard to predict branches, which guard independent memory accesses that often miss in the d-cache. Delayed Fetch is the optimal mechanism for *astar* because it is able to bring more of these independent memory accesses into the instruction window than Dynamic Predication. Furthermore, Delayed Fetch eliminates 80% of the total branch mispredictions (Figure 5.6.

Deepsjeng_17 is another good fit for Delayed Fetch, however here the overheads associated with Delayed Fetch (injecting MOV instructions) waste more fetch bandwidth than the branch mispredictions did. This leads the benchmarks to lose a small amount of performance overall. Again, the waste-based throttling mechanism should eliminate this negative result; however, due to the throttling mechanism not being perfect this workload results in a performance loss.

Other workloads like *xz_17* and *sjeng_06* suffer from an increase in the critical path due to Delayed Fetch increasing the latency of correct-path instructions. In the future work section, we discuss the need for a critical path predictor that can identify such cases where Delayed Fetch leads to performance inversions.

Combining Dynamic Predication and Delayed Fetch. While many improvements must be made to both Dynamic Predication and Delayed Fetch, it is already apparent that combination of alternatives is better than either individually. In this dissertation, we consider the simplest switching mechanism: switching at the application granularity. Switching is most valuable for workloads like *mcf_06* that are a good fit for one mechanism (Dynamic Predication), but a bad fit for another (Delayed Fetch). Ideally, switching would happen at a finer granularity within the application. This concept is discussed more in the future work section of this chapter.

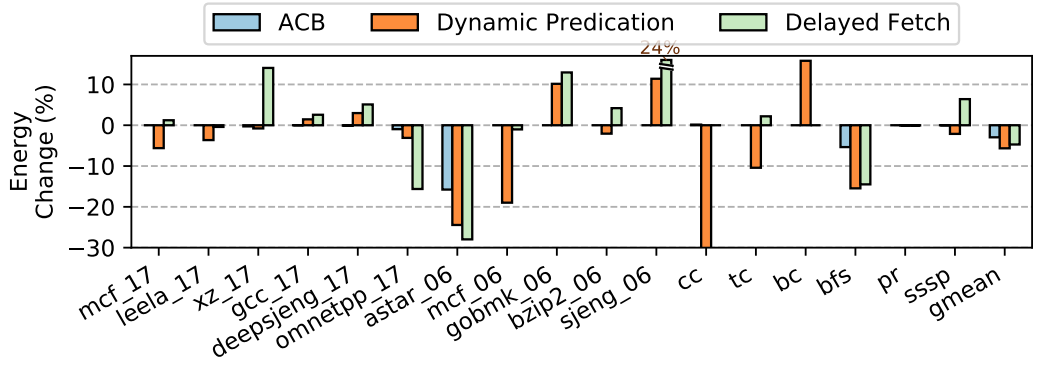

Figure 5.8: Change in Energy for Dynamic Predication and Delayed Fetch.

5.5.4 Energy Results

Figure 5.8 shows the change in energy for both Dynamic Predication and Delayed Fetch. These results largely reflect the performance results from the previous section due to two reasons. First, the static (i.e., leakage) energy, which McPAT estimates as roughly 50% of the total energy, increases or decreases linearly with the time it takes to complete the program. Therefore the increase in static energy will reflect the change in time, i.e., the change in performance. The dynamic energy is primarily effected by two variables: 1) the change in time to complete the program and 2) the change in activity factor. As was the case for static energy, the dynamic energy will be scaled up or down linearly with respect to performance. However, the dynamic energy will further be affected by the change in activity factor, which is mainly impacted by the change in issued micro-ops.

Figure 5.9 shows the reduction in issued micro-ops (i.e., higher bar

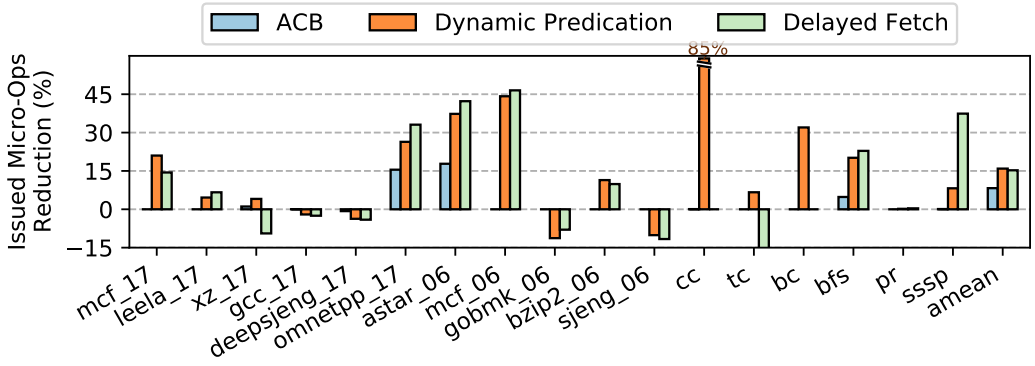

Figure 5.9: Reduction in total micro-ops issued to the execution units for Dynamic Predication and Delayed Fetch.

equates to fewer micro-ops issued). As the results show, both Dynamic Predication and Delayed Fetch can lead to fewer micro-ops issued, which in turn lowers dynamic energy. Dynamic Predication and Delayed Fetch may reduce the total number of issued micro-ops by avoiding branch mispredictions, which will in turn avoid fetching micro-ops down the wrong-path. Some of these positive effects are reduced by the overheads of Dynamic Predication (wrong-path predicated ops and CMOV ops) and Delayed Fetch (MOV ops). These overheads become significant in cases where the branch was predicted correctly a high percentage of the time. The Hard Branch Table may detect such branches because their misprediction rate (i.e., misprediction per kilo instruction) still exceeds the given threshold; however, a branch with a high MPKI can still be predicted correctly a large majority of the time. Unfortunately, Dynamic Predication and Delayed fetch transform the branch each time it is fetched, meaning that the additional MOV ops and wrong-path predicated ops are

waste in cases when the branch would have been predicted correctly. The future work section in this chapter discusses ways to reduce the overheads of these techniques in cases where this imbalance exists, which should continue to improve dynamic energy.

5.6 Related Work

Wish Branches [29] implemented Dynamic Predication by having the compiler generate both the predicated and non-predicated versions of the original code. The hardware would the monitor branch predictor performance and dynamically select which version of the code should be used. While this approach can be useful in some cases, it still ultimately relies on the compiler to determine which branches should be predicated. Further, the compiler may need to generate many different combinations of the code to account for all cases where a branch should or should not be predicated. Our implementation of Dynamic Predication solves this by detecting which branches are hard-to-predict and should be predicated. Furthermore, we dynamically predict the location of the merge point, eliminating the need for compiler or ISA support for Dynamic Predication.

Transparent Control Independence (TCI) proposes a stack-based method for tracking nested merge points similar to the one used in this dissertation [6]. Our implementation, however, has the small simplification that all nested branches inherit the same merge point prediction for their parent.

5.7 Future Work

Clearly, both Dynamic Predication and Delayed Fetch still have room to improve before they are beneficial for all applications. In this section, I propose several techniques which I think will broaden the cases where Dynamic Predication and Delayed Fetch are effective, while also eliminating the negative outlier cases.

5.7.1 Eliminating MOV micro-ops

Move micro-ops are a considerable overhead for both Dynamic Predication and Delayed Fetch. The primary purpose of the move micro-ops is to patch the data-flow graph so that the proper data dependencies are used after branch resolution. Move micro-ops are the simplest solution; however, given their high overhead, more complex solutions should also be explored. Modifications to the rename hardware to reuse tags and the broadcast hardware, to delay the broadcast of tags, may be possible which negate the need for move micro-ops to consume fetch/execution resources.

5.7.2 Critical Path Based Throttling

Currently, Dynamic Predication and Delayed Fetch used a Waste-Based throttling technique to throttle branches where either Predication or Delayed Fetch is more expensive than the branch mispredictions themselves. However, waste is only half of the story. Both Dynamic Predication and Delayed Fetch can have negative effects on the critical path of the application. Fields et

al. [17] developed a token-passing critical path detection algorithm used to detect which instructions are on the critical path of a program. This token-passing algorithm can be modified to detect whether or not specific paths effected by Dynamic Predication or Delayed Fetch resulted in changes to the critical path. This information can be used to further inform the switching algorithm to choose the mechanism that minimizes the critical path. This mechanism would be particularly beneficial for Delayed Fetch, whose main drawback is increasing the latency of correct-path instructions.

5.7.3 Utilizing TAGE confidence

In the current design, the Hard Branch Table (HBT) is used to determine which static branches mispredict enough to justify using either Dynamic Predication or Delayed Fetch. Furthermore, the HBT tracks which branches flush the most micro-ops from the ROB as a mechanism for detecting which branch mispredictions have the highest impact on performance. As discussed in Chapter 2, some branches are identified due to their long execution latency rather than their high misprediction rate. In extreme cases, branches with a relatively low misprediction rate (10%) can be identified for Dynamic Predication or Delayed Fetch. This is not ideal, however, because our existing mechanism will attempt to Predicate or Delay Fetch *all* dynamic instances of the branch. Ideally, we would like to only Predicate or Delay Fetch the instances of the branch that are likely to mispredict and continue to use branch prediction for the cases that are likely to be correct. Fortunately, prior work

is capable of identifying such cases when the branch predictor is confident or not-confident with no storage overhead [51]. This confidence mechanism can be used to enable Dynamic Predication or Delayed Fetch dynamically, which creates opportunity to use the branch predictor when it is confident.

Unfortunately, both Dynamic Predication and Delay Fetch require the code to be transformed to the *predicated order*, which in turn requires a warmup period for the branch predictor. Dynamically switching between branch prediction and Dynamic Predication/Delayed Fetch would therefore be harmful as the branch predictor would never be given the opportunity to warm up. As a solution to this, I propose maintaining the predicated order, even when using branch prediction, for such branches. This will allow the branch predictor to warm up, which will in turn allow us to detect the confidence of the branch predictor. By maintaining the core's view of the code, we can switch back and forth between Dynamic Predication/Delayed Fetch and branch prediction, thus allowing us to use each mechanism only when it will be most beneficial.

5.7.4 Specialized Loop Predictor

Variable length loops are a common form of hard-to-predict branch. Such loops are not predicted accurately by the Loop component of the TAGE-SC-L predictor due to the fact that the loop variable is constantly changing. Further, both the merge point predictor presented in this dissertation and prior work in merge point prediction [15] predict sub-optimal merge points for most

loops, which results in performance inversions when Dynamic Predication or Delayed Fetch are applied. The minimum/maximum iteration counts, however, are often very stable values for most variable length loops. This creates an opportunity to predict the minimum and maximum iteration count of a variable length loop. With this information, Dynamic Predication or Delayed Fetch could be applied to each loop iteration between the minimum and maximum iteration count. For variable loops that have a low variance in iteration count, this could substantially improve performance.

Chapter 6

Conclusion

This dissertation presents new microarchitecture techniques that are able to pre-compute or avoid branch mispredictions in cases where TAGE-SC-L, the current state-of-the-art branch predictor, cannot. Each technique proposed in this dissertation is enabled by a new merge point prediction algorithm that significantly improves accuracy and coverage compared to prior work. The accurate merge point predictor enables the design of Branch Runahead by enabling the microarchitecture to detect control and data dependencies between dependence chains (i.e., affector and guard branches). Without the detection of these important dependencies, Branch Runahead would not be able to accurately run as far ahead, which would reduce the accuracy and timeliness of the predictions it generates. For Dynamic Predication and Delayed Fetch, knowing the location of the merge point enables these techniques to avoid predicting the direction of the branch at all. Instead, we are able to either fetch both paths of the branch, or fetch neither, based on what is most appropriate for the code. Clearly, improving the coverage of the merge point predictor enables these techniques to be used on more hard to predict branches.

This dissertation proposes a total of three alternatives to branch prediction. Each alternative comes with its own set of trade-offs that make it more or less appropriate in different situations. Branch Runahead performs best for data-dependent branches with short dependence chains. Fortunately, this happens to be a category of hard-to-predict branch that traditional history-based branch predictors have struggled with. Branch Runahead is able to quickly and efficiently execute the short dependence chain to pre-compute the direction of the branch. Branch Runahead also uses a predictive initiation strategy, which allows it to run multiple dependence chains in parallel. Predictive initiation enables high chain-level parallelism, which in turn enables Branch Runahead to deliver predictions in a more timely way. Branch Runahead differs from prior work in three primary ways: 1) dependence chains are extracted at runtime, which enables Branch Runahead to 2) identify affector/guard branches as well as biased branches at runtime. Detecting these cases at runtime is more accurate than identifying them at compile time, which makes Branch Runahead's dependence chains lighter-weight than those produced by a compiler. Finally, Branch Runahead uses the affector/guard branch information to 3) execute dependence chains continuously. This enables Branch Runahead to run farther ahead than prior light-weight runtime-only techniques.

Dynamic Predication and Delayed Fetch, on the other hand, work best when both traditional branch prediction and Branch Runahead have failed. These options are a last resort, which improve performance by eliminating branch misprediction. This dissertation identifies the reciprocal trade-offs

between the two techniques. This motivates the design of a single Control Independent Microarchitecture that is capable of dynamically switching between each technique. While further research is still required to handle performance inversion cases, both Dynamic Predication and Delayed Fetch show high potential to significantly improve performance and energy in the presence of impossible-to-predict branches. This work significantly reduces total micro-ops executed without significantly impacting performance, which in turn lowers dynamic power.

Together, these three techniques improve performance for many branches which state-of-the-art branch predictors cannot handle. As history-based branch predictors continue to struggle with the remaining hard-to-predict branches, the techniques presented in this dissertation will be required to continue to improve performance. This dissertation provides a framework for the microarchitecture to identify such branches, then switch to a new mechanism that best fits the needs of the branch.

Bibliography

[1] "Nsf/intel partnership on foundational microarchitecture research (fomr)," https://www.nsf.gov/pubs/2019/nsf19598/nsf19598.htm.

[2] "Scarab," https://github.com/hpsresearchgroup/scarab.

[3] "The standard performance evaluation corporation (spec). the spec benchmark suite," http://www.spec.org.

[4] H. Akkary, S. Srinivasan, R. Koltur, Y. Patil, and W. Refaai, "Perceptron-based branch confidence estimation," in 10th International Symposium on High Performance Computer Architecture (HPCA'04), 2004, pp. 265–265.

[5] M. Al-Otoom, E. Forbes, and E. Rotenberg, "Exact: Explicit dynamic-branch prediction with active updates," in Proceedings of the 7th ACM International Conference on Computing Frontiers, ser. CF '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 165–176. [Online]. Available: https://doi.org/10.1145/1787275.1787321

[6] A. S. Al-Zawawi, V. K. Reddy, E. Rotenberg, and H. H. Akkary, "Transparent control independence (tci)," in Proceedings of the 34th Annual International Symposium on Computer Architecture, ser. ISCA

'07. New York, NY, USA: ACM, 2007, pp. 448–459. [Online]. Available: http://doi.acm.org/10.1145/1250662.1250717

[7] G. Ayers, H. Litz, C. Kozyrakis, and P. Ranganathan, "Classifying memory access patterns for prefetching," in Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 513–526. [Online]. Available: https://doi.org/10.1145/3373376.3378498

[8] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP benchmark suite," CoRR, vol. abs/1508.03619, 2015. [Online]. Available: http://arxiv.org/abs/1508.03619

[9] T. E. Carlson, W. Heirman, O. Allam, S. Kaxiras, and L. Eeckhout, "The load slice core microarchitecture," in Proceedings of the 42nd Annual International Symposium on Computer Architecture, ser. ISCA '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 272–284. [Online]. Available: https://doi-org.ezproxy.lib.utexas.edu/10.1145/2749469.2750407

[10] R. S. Chappell, F. Tseng, A. Yoaz, and Y. N. Patt, "Difficult-path branch prediction using subordinate microthreads," in Proceedings 29th Annual International Symposium on Computer Architecture, May 2002, pp. 307–317.

[11] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt, "Simultaneous subordinate microthreading (ssmt)," in Proceedings of the 26th Annual International Symposium on Computer Architecture, ser. ISCA '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 186–195. [Online]. Available: http://dx.doi.org/10.1145/300979.300995

[12] A. Chauhan, J. Gaur, Z. Sperber, F. Sala, L. Rappoport, A. Yoaz, and S. Subramoney, "Auto-predication of critical branches*," in 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), 2020, pp. 92–104.

[13] C.-Y. Cher and T. N. Vijaykumar, "Skipper: a microarchitecture for exploiting control-flow independence," in Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34, Dec 2001, pp. 4–15.

[14] Y. Chou, J. Fung, and J. P. Shen, "Reducing branch misprediction penalties via dynamic control independence detection," in Proceedings of the 13th International Conference on Supercomputing, ser. ICS '99. New York, NY, USA: ACM, 1999, pp. 109–118. [Online]. Available: http://doi.acm.org/10.1145/305138.305175

[15] J. D. Collins, D. M. Tullsen, and H. Wang, "Control flow optimization via dynamic reconvergence prediction," in Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on, Dec 2004, pp. 129–140.

[16] M. U. Farooq, Khubaib, and L. K. John, "Store-load-branch (slb) predictor: A compiler assisted branch prediction for data dependent branches," in 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA), 2013, pp. 59–70.

[17] B. Fields, S. Rubin, and R. Bodik, "Focusing processor policies via critical-path prediction," in Proceedings 28th Annual International Symposium on Computer Architecture, 2001, pp. 74–85.

[18] H. Gao, Y. Ma, M. Dimitrov, and H. Zhou, "Address-branch correlation: A novel locality for long-latency hard-to-predict branches," in 2008 IEEE 14th International Symposium on High Performance Computer Architecture, 2008, pp. 74–85.

[19] M. F. Gary Tyson, Kelsey Lick, "Limited Dual Path Execution," Michigan, Research Report CSE-TR-346-97, 1997. [Online]. Available: https://www.eecs.umich.edu/techreports/cse/1997/CSE-TR-346-97.pdf

[20] D. Grunwald, A. Klauser, S. Manne, and A. Pleszkun, "Confidence estimation for speculation control," in Proceedings of the 25th Annual International Symposium on Computer Architecture, ser. ISCA '98. USA: IEEE Computer Society, 1998, p. 122–131. [Online]. Available: https://doi.org/10.1145/279358.279376

[21] S. Gupta, N. Soundararajan, R. Natarajan, and S. Subramoney, "Opportunistic early pipeline re-steering for data-dependent branches,"

in Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques, ser. PACT '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 305–316. [Online]. Available: https://doi-org.ezproxy.lib.utexas.edu/10.1145/3410463.3414628

[22] M. Hashemi, O. Mutlu, and Y. N. Patt, "Continuous runahead: Transparent hardware acceleration for memory intensive workloads," in The 49th Annual IEEE/ACM International Symposium on Microarchitecture, ser. MICRO-49. Piscataway, NJ, USA: IEEE Press, 2016, pp. 61:1–61:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=3195638.3195712

[23] M. Hashemi and Y. N. Patt, "Filtered runahead execution with a runahead buffer," in Proceedings of the 48th International Symposium on Microarchitecture, ser. MICRO-48. New York, NY, USA: ACM, 2015, pp. 358–369. [Online]. Available: http://doi.acm.org/10.1145/2830772.2830812

[24] H. Heiss and E. Wallmeier, "Applications of the leaky bucket throughput characterisation," in IEEE ATM '97 Workshop Proceedings (Cat. No.97TH8316), 1997, pp. 195–203.

[25] A. D. Hilton and A. Roth, "Ginger: Control independence using tag rewriting," in Proceedings of the 34th Annual International Symposium on Computer Architecture, ser. ISCA '07. New York,

NY, USA: ACM, 2007, pp. 436–447. [Online]. Available: http://doi.acm.org/10.1145/1250662.1250716

[26] E. Jacobsen, E. Rotenberg, and J. E. Smith, "Assigning confidence to conditional branch predictions," in Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture, ser. MICRO 29. Washington, DC, USA: IEEE Computer Society, 1996, pp. 142–152. [Online]. Available: http://dl.acm.org/citation.cfm?id=243846.243880

[27] D. A. Jimenez, "Composite confidence estimators for enhanced speculation control," in 2009 21st International Symposium on Computer Architecture and High Performance Computing, 2009, pp. 161–168.

[28] H. Kim, J. A. Joao, O. Mutlu, and Y. N. Patt, "Diverge-merge processor (dmp): Dynamic predicated execution of complex control-flow graphs based on frequently executed paths," in Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, ser. MICRO 39. Washington, DC, USA: IEEE Computer Society, 2006, pp. 53–64. [Online]. Available: https://doi.org/10.1109/MICRO.2006.20

[29] H. Kim, O. Mutlu, J. Stark, and Y. N. Patt, "Wish branches: Combining conditional branching and predication for adaptive predicated execution," in Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture, ser. MICRO 38. USA: IEEE Computer Society, 2005, p. 43–54. [Online]. Available: https://doi-org.ezproxy.lib.utexas.edu/10.1109/MICRO.2005.38

[30] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," IEEE Computer Architecture Letters, vol. 15, no. 1, pp. 45–49, 2016.

[31] S. Kondguli and M. Huang, "R3-dla (reduce, reuse, recycle): A more efficient approach to decoupled look-ahead architectures," in 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA), Feb 2019, pp. 533–544.

[32] M. S. Lam and R. P. Wilson, "Limits of control flow on parallelism," in [1992] Proceedings the 19th Annual International Symposium on Computer Architecture, 1992, pp. 46–57.

[33] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in 2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2009, pp. 469–480.

[34] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, ser. PLDI '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 190–200. [Online]. Available: https://doi.org/10.1145/1065010.1065034

[35] S. Mahlke, R. Hank, R. Bringmann, J. Gyllenhaal, D. Gallagher, and W.-M. Hwu, "Characterizing the impact of predicated execution on branch prediction," in Proceedings of MICRO-27. The 27th Annual IEEE/ACM International Symposium on Microarchitecture, 1994, pp. 217–227.

[36] K. Malik, M. Agarwal, V. Dhar, and M. I. Frank, "Paco: Probability-based path confidence prediction," in 2008 IEEE 14th International Symposium on High Performance Computer Architecture, 2008, pp. 50–61.

[37] L. Meng and S. Oyanagi, "Control independence using dual renaming," in 2010 First International Conference on Networking and Computing, Nov 2010, pp. 264–267.

[38] P. Michaud, "An alternative tage-like conditional branch predictor," ACM Trans. Archit. Code Optim., vol. 15, no. 3, pp. 30:1–30:23, Aug. 2018. [Online]. Available: http://doi.acm.org/10.1145/3226098

[39] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: An effective alternative to large instruction windows," IEEE Micro, vol. 23, no. 6, pp. 20–25, Nov 2003.

[40] A. Naithani, J. Feliu, A. Adileh, and L. Eeckhout, "Precise runahead execution," in 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2020, pp. 397–410.

[41] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using simpoint for accurate and efficient simulation," in

Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, ser. SIGMETRICS '03. New York, NY, USA: ACM, 2003, pp. 318–319. [Online]. Available: http://doi.acm.org/10.1145/781027.781076

[42] N. Premillieu and A. Seznec, "Syrant: Symmetric resource allocation on not-taken and taken paths," ACM Trans. Archit. Code Optim., vol. 8, no. 4, pp. 43:1–43:20, Jan. 2012. [Online]. Available: http://doi.acm.org/10.1145/2086696.2086722

[43] N. Prémillieu and A. Seznec, "SPREPI: Selective Prediction and REplay for predicated Instructions," INRIA, Research Report RR-8351, Aug. 2013. [Online]. Available: https://hal.inria.fr/hal-00856160

[44] S. Pruett and Y. Patt, "Dynamic merge point prediction," 2020. [Online]. Available: https://arxiv.org/abs/2005.14691

[45] S. Pruett and Y. Patt, "Branch runahead: An alternative to branch prediction for impossible to predict branches," in MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 804–815. [Online]. Available: https://doi.org/10.1145/3466752.3480053

[46] G. Reinman, T. Austin, and B. Calder, "A scalable front-end architecture for fast instruction delivery," in Proceedings of the 26th

Annual International Symposium on Computer Architecture, ser. ISCA '99. USA: IEEE Computer Society, 1999, p. 234–245. [Online]. Available: https://doi.org/10.1145/300979.300999

[47] E. Rotenberg, Q. Jacobson, and J. Smith, "A study of control independence in superscalar processors," in Proceedings Fifth International Symposium on High-Performance Computer Architecture, Jan 1999, pp. 115–124.

[48] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith, "Trace processors," in Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture, ser. MICRO 30. Washington, DC, USA: IEEE Computer Society, 1997, pp. 138–148. [Online]. Available: http://dl.acm.org/citation.cfm?id=266800.266814

[49] A. Roth and G. S. Sohi, "Speculative data-driven multithreading," in Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture, Jan 2001, pp. 37–48.

[50] A. Roth and G. Sohi, "Squash reuse via a simplified implementation of register integration," Journal of Instruction-Level Parallelism, vol. 3, pp. l–7, 2002. [Online]. Available: https://www.jilp.org/vol3/

[51] A. Seznec, "Storage free confidence estimation for the tage branch predictor," in 2011 IEEE 17th International Symposium on High Performance Computer Architecture, Feb 2011, pp. 443–454.

[52] A. Seznec, "TAGE-SC-L Branch Predictors," in JILP - Championship Branch Prediction, Minneapolis, United States, Jun. 2014. [Online]. Available: https://hal.inria.fr/hal-01086920

[53] A. Seznec, "Exploring branch predictability limits with the mtage+sc predictor," in 5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5), 2016.

[54] R. Sheikh, J. Tuck, and E. Rotenberg, "Control-flow decoupling: An approach for timely, non-speculative branching," IEEE Transactions on Computers, vol. 64, no. 8, pp. 2182–2203, Aug 2015.

[55] J. E. Smith, "A study of branch prediction strategies," in Proceedings of the 8th Annual Symposium on Computer Architecture, ser. ISCA '81. Washington, DC, USA: IEEE Computer Society Press, 1981, p. 135–148.

[56] A. Sodani and G. S. Sohi, "Dynamic instruction reuse," in Proceedings of the 24th Annual International Symposium on Computer Architecture, ser. ISCA '97. New York, NY, USA: ACM, 1997, pp. 194–205. [Online]. Available: http://doi.acm.org/10.1145/264107.264200

[57] V. Srinivasan, R. B. R. Chowdhury, and E. Rotenberg, "Slipstream processors revisited: Exploiting branch sets," in 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), 2020, pp. 105–117.

[58] J. W. Stark, Out-of-order fetch, decode, and issue. Ph.D. Dissertation, University of Michigan, 2000.

[59] K. Sundaramoorthy, Z. Purser, and E. Rotenberg, "Slipstream processors: Improving both performance and fault tolerance," SIGPLAN Not., vol. 35, no. 11, pp. 257–268, Nov. 2000. [Online]. Available: http://doi.acm.org/10.1145/356989.357013

[60] K. K. Sundararaman and M. Franklin, "Multiscalar execution along a single flow of control," in Proceedings of the 1997 International Conference on Parallel Processing (Cat. No.97TB100162), Aug 1997, pp. 106–113.

[61] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2sim: A simulation framework for cpu-gpu computing," in Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, ser. PACT '12. New York, NY, USA: ACM, 2012, pp. 335–344. [Online]. Available: http://doi.acm.org/10.1145/2370816.2370865

[62] S. Zangeneh, S. Pruett, S. Lym, and Y. N. Patt, "Branchnet: A convolutional neural network to predict hard-to-predict branches," in 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2020, pp. 118–130.

[63] C. B. Zilles and G. S. Sohi, "Understanding the backward slices of performance degrading instructions," in Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201), June 2000, pp. 172–181.

[64] C. Zilles and G. Sohi, "Execution-based prediction using speculative slices," SIGARCH Comput. Archit. News, vol. 29, no. 2, pp. 2–13, May 2001. [Online]. Available: http://doi.acm.org/10.1145/384285.379246