UNIVERSITY OF CALIFORNIA
SANTA CRUZ

LOAD DRIVEN BRANCH PREDICTOR (LDBP)

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER ENGINEERING

by

Akash Sridhar

March 2021

The Dissertation of
Akash Sridhar is approved:

Professor Jose Renau, Chair

Professor Anujan Varma

Professor Nicholas Brummell

Quentin Williams
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

Load Driven Branch Predictor (LDBP)

by

Akash Sridhar

A larger instruction window on Out-of-Order (OoO) cores facilitates better exploitation of inherent Instruction Level Parallelism (ILP). Branch miss-speculation penalty restricts scaling to larger instruction window in OoO cores. Branch instructions dependent on hard-to-predict load data are the leading misprediction contributors. Computer architects continuously strive to optimize branch prediction algorithms and increase predictor size to mitigate mispredictions. Current state-of-the-art history-based branch predictors have low prediction accuracy for these branches. Prior research backs this observation by showing that increasing the size of a 256-KBit history-based branch predictor to its 1-MBit variant has just a 10% reduction in branch mispredictions.

In this dissertation, I present the novel Load Driven Branch Predictor (LDBP), specifically targeting hard-to-predict branches dependent on a load instruction. Though random load data determines these branches' outcomes, most of these data's load address have a predictable pattern. This is an observable template in data structures like arrays and maps. The LDBP predictor model exploits this behavior to trigger future loads associated with branches ahead of time and use its data to predict its outcome. The predictable loads are tracked, and the branch instruction's precomputed outcomes are buffered for making predictions. The experimental results show that on a modern Zen2-like OoO core, compared to a standalone 256-Kbit IMLI

predictor, when LDBP is augmented to it, the average branch mispredictions reduce by 12% and the average IPC improves 7.14% for benchmarks from SPEC CINT2006 and GAP benchmark suite.

Acknowledgments

I would not be writing this dissertation now without the support and encouragement of my mom (Meena), dad (Sridhar), and sister (Dr. Archana Sridhar). Without them, I would not have garnered the courage to get out of my hometown's comforts and pursue my Ph.D. in the US. As I wrote this dissertation, I realized that both Meena and Sridhar's children are Doctors (though my sister is the real DOCTOR). A special thanks to Vignesh Maheswaran (my brother-in-law) for his non-stop entertainment, which played a solid role in keeping me sane during this journey.

Prof. Venkateswaran Nagarajan (fondly addressed as Prof. Waran) played an integral part in shaping my life and career. During the initial years of my undergrad degree in Chennai, he made me realize my academic potential. Without his guidance, I would never have even thought of doing a Ph.D. He made me realize that I was capable of taking the road less traveled. Words are not enough to highlight Prof. Waran's contribution. Thank you, Sir.

Hamsini Sankaran is a late entrant into my life. I knew her when I was six years old, but I never realized that she would be such an essential part of my life. It took me 22 more years to understand that. The journey of obtaining a Ph.D. is never straightforward. There were testing moments towards the end of that journey, and there were times I thought I should quit and get out with a Master's degree. You held me together and made sure I did not lose my confidence.

I want to thank my lab mates at the MASC research group at UCSC - Sina, Rafael, Blake, Daphne, Sakshi, Nilufar, and Gabriel. I appreciate all your help and support during my Doctoral degree. A special thanks to Sheng and Nursultan. I had the most collaborations with the two of you, and I cannot thank you enough for your valuable contributions and feedbacks to

my thesis. I want to send a special shoutout to Rohan for being a fantastic friend. You made sure I never missed Chennai.

Dr. Eshwar Chandrasekharan and I share a special bond. He is my best friend from primary school, and I am happy both our Ph.D. programs overlapped during the same time frame. This gave me a fantastic opportunity to talk to him about research, industry trends, and stories from our dreaded pasts. Thanks, macha.

I am delighted to acknowledge Janani Ganesan's contribution to my Ph.D. degree. She is one of the very few persons I shared every high and low during my Ph.D. program. She made sure I never let my guard down, and I always stayed focused on my ultimate goal. She has contributed so much to my physical health and mental well-being. She taught me amazing cooking recipes, and she's my workout sensei.

I am saving my biggest thanks to Prof. Jose Renau. I always count my blessings for getting an advisor like Jose. Right from the start, he has been super supportive and ready to guide me throughout this journey. He gave me the space to explore new avenues, which helped me grow as a researcher. Whatever I am now, I owe it entirely to Jose. I will forever cherish my time at the MASC research lab.

Chapter 1

Introduction

Imagination is more important than knowledge. For knowledge is limited, whereas imagination embraces the entire world, stimulating progress, giving birth to evolution.

Albert Einstein

1.1 Introduction

The clock cycle time, instruction count, and the average cycles required to execute an instruction(CPI) directly influence the performance of a processor [48]. Over the past few decades, architects focused on increasing the processor clock frequency and scaling resource size to improve the overall performance. To support faster cycle time, pipelines became deeper to bridge the gap between the clock speed and the delay of different micro-architectural components.

Deeper pipelines contribute to increased clock skew and performance degradation due to longer stalls and hazards. The leakage power also inflates with smaller transistors and faster clock speed. The insignificant clock speed gains restricts the scaling of hardware resources in a core.

To maximize the processor's performance potential and counter the effect of increasing thermal limitations and the restrictions on single-core scalability, architects increased the core count [12, 36]. Multi-cores intends to keep up the performance scaling proportional to the increase in transistor density. Many researchers are focusing on designing chips with several hundred cores. Parallel workloads exploit the benefits of multi-cores to the fullest. Multi-cores were able to achieve a considerable increase in performance compared to their single-core counterparts by exploiting the Thread-Level Parallelism (TLP) of applications and through the support of additional architectural resources. However, most general-purpose applications have an insignificant amount of TLP [6]. Thus, increasing the core count does not yield a solution to address the performance limitations of low TLP applications.

An alternate approach would be to cash in on the inherent Instruction Level Parallelism (ILP) within each thread. Speeding up each thread improves the overall performance. Increasing ILP comes at a cost - the microarchitecture must have wider instruction window, better prefetching, near-perfect branch prediction, higher number of physical register, more functional units and so on. However, exploiting higher ILP by issuing more instructions per cycle involves increased hardware complexity, which in turn has implications on the cycle time and energy dissipation. For example, with higher issue width, more functional units will be added to support the higher inflow of instructions. A minimal increase in issue width involves the accrue of much additional hardware which contributes to increased complexity and power consumption. This

also swells the number of interconnections wires between these components, contributing to increased wire delay. The wire delay adds to a rise in the overall delay, thereby curbing the increase of clock frequency [27]. If the complexity of an architectural component on the critical path of the processor increases, it might add additional clock cycles to the critical path delay.

Branch mispredictions is a significant factor limiting single-thread performance in modern microprocessors [24]. The miss penalty incurred due to incorrect branch prediction is very high [46]. Along with data cache misses, incorrect branch prediction causes significant performance degradation. Optimizing branch predictors does not involve too much overhead on the pipeline's critical path. Minor improvements to branch prediction accuracy can yield significant performance gains. Improving the branch prediction accuracy has several benefits. First, it improves IPC by reducing the number of flushed instructions. Second, it reduces the power dissipation incurred through the execution of instructions taking the wrong path of the branch. Third, it increases the Memory Level Parallelism (MLP), which facilitates a deeper instruction window in the pipeline and supports multiple outstanding memory operations.

1.1.1 What is the problem with existing branch predictor models?

Current branch prediction championships use either perceptron-based predictors [18, 19, 20, 21] or TAGE-based predictors [39, 41]. These predictors may use global and local history, and a statistical corrector to further improve performance. The TAGE-SC-L [44], which is a derivative of its previous implementation from Championship Branch Prediction (CBP-4) [42], combined several of these techniques and was the winner of the last branch prediction championship (CBP-5). Numbers from CBP-5 [43, 44] shows that scaling from a 64-Kbit TAGE

predictor to unlimited size, only yields branch Mispredictions per Kilo Instructions (MPKI) reduction from 3.986 to 2.596.

Most of the current processors like AMD Zen 2/3, ARM A72-A77 and Intel Skylake use some TAGE variation branch predictor. TAGE-like predictors are excellent, but there are still many difficult-to-predict branches. Seznec [42, 44] studied the prediction accuracy of a 256-Kbit TAGE predictor and a no storage limit TAGE. The 256-Kbit TAGE had only about 10% more mispredictions than its infinite size counterpart. The numbers mentioned above would reflect the prediction accuracy of the latest Zen 2 CPU [51] using a 256-Kbit TAGE-based predictor. For this work, the 256-Kbit TAGE-GSC + IMLI [47] is used, which combines the global history components of the TAGE-SC-L with a loop predictor and local history as the baseline system.

Recent work [26] shows that even though the current state-of-the-art branch predictors have almost perfect prediction accuracy, there is scope for gaining significant performance by fixing the remaining mispredictions. The core architecture could be tuned to be wider if it had the support of better branch prediction, which could potentially offer more IPC gains. Prior works [9, 11] have tried to address different types of hard-to-predict branches. A vital observation of these works is that most branches that state-of-the-art predictors fail to capture are branches that depend on a recent load. If the data loaded is challenging to predict, TAGE-like predictors have a low prediction accuracy as these patterns are arbitrary and too large to be captured.

To showcase the extent of speedup exploitable with optimal branch prediction, the IPC gains of an AMD Zen 2 core having oracle (prefect) branch prediction is compared to a same core having a 256-Kbit IMLI predictor. Table 1.1 shows that perfect branch prediction can achieve an average IPC gain of 50.85% across the different benchmarks tested. The oracle

configuration yielded this IPC improvement without any architectural optimizations to the Zen 2 architecture. This architecture could be tuned to be wider if it had the support of better branch prediction, which could potentially offer more IPC gains. These numbers clearly show that even though current state-of-the-art branch predictors offer very high prediction accuracy (in the range of 95% to 99%), there is still tremendous scope to improve IPC gains by fixing the remaining mispredictions. As discussed earlier in this chapter, doing so will also result in a more energy-efficient core.

Benchmark	**% IPC Increase**
spec06_xalan	5.07
spec06_sjeng	14.61
spec06_perlbench	3.52
spec06_omnetpp	4.03
spec06_mcf	80.99
spec06_libquantum	0.86
spec06_hmmer	81.82
spec06_h264ref	3.99
spec06_gobmk	33.55
spec06_gcc	3.256
spec06_bzip2	14.91
spec06_astar	91.21
gap_tc	103.74
gap_sssp	66.29
gap_pr	6.67
gap_cc	170.59
gap_bfs	173.85
gap_bc	56.41

Table 1.1: Percentage increase in IPC running Oracle branch predictor against baseline 256-Kbit IMLI. Fixing hard-to-predict branches yield tremendous IPC gains.

1.1.2 Contribution of the Dissertation

The critical observation/contribution of this thesis is that although the load data feeding a load-dependent branch may be random, the load address may be predictable. If the branch operand(s) are dependent on arbitrary load data, the branch is going to be difficult to predict. If the load address is predictable, it is possible to ”prefetch” the load ahead of time, and use the actual data value in the branch predictor.

Based on the previous observation, I propose to combine the stride address predictor [10] with a new type of the branch predictor to trigger future loads ahead of time and feed the load data to the branch predictor. The trigger load data is used to pre-compute the associated dependent branch. Then, when the corresponding branch gets fetched, the proposed predictor will have a very high accuracy even with random data. The predictor is only active for branches that have low confidence with the default predictor and depends on loads with predictable addresses. Otherwise, the default IMLI predictor performs the prediction. The proposed predictor is called Load Driven Branch Predictor (LDBP).

LDBP is an implementation of a new class of branch predictors that combine load(s) and branches to perform prediction. This new class of load-assisted branch predictors allows having near-perfect branch prediction accuracy over random data as long as the load address is predictable. It is still a prediction because there are possibilities of coherence or other forwarding issues that can make it difficult to guarantee the results.

LDBP does not require software changes or modifications to the ISA. It tracks the backward code slice starting from the branch and terminating at a set of one or more loads. If

```
1 addi     a5,a5,4 //increments array index
2   .
3   .
4   .
5   .
6 lw       a4,0(a5)//loads data from array
7 bnez     a4,1043e <main+0x44>
```

Figure 1.1: Vector traversal code snippet example, *bnez* in line 5 is the most mispredicting branch in this kernel

all the loads have a predictable address, and the slice is small enough to be computed, LDBP keeps track of the slice. When the same branch retires again, it will start to trigger future loads ahead of time. The future fetches of this branch uses their corresponding trigger load(s) data to pre-compute the slice result and determine the branch outcome. Through the rest of this theis, the load (with predictable address) that has a dependency with a branch will be referred as a trigger load and its dependent branch will be referred as a load-dependent branch.

I will explain a simple code example that massively benefits from LDBP. Let us consider a simple kernel that iterates over a vector having random 0s and 1s to find values greater than zero. The branch with most mispredictions in this kernel has the assembly sequence shown in Figure 1.1. As traversal is done over a vector, the load addresses here are predictable, even though the data is completely random. TAGE fails to build these branch history patterns due to the dependence of the branch outcome on irregular data patterns. LDBP has near-perfect branch prediction because the trigger load (line 4) has a predictable address. LDBP triggers loads ahead of time, computes the branch-load backward slice, and stores the results. The branch uses the precomputed outcome at fetch. When LDBP is augmented to a Zen 2 like core with a 256-Kbit IMLI predictor, the IPC improves by 4.1x times.

In general, a load-dependent branch immediately follows a trigger load in program order. Due to the narrow interval between these two instructions, the load data will not be available when the branch is fetched. The key challenge is to send trigger loads in a timely manner before its corresponding load-dependent branch is fetched. LDBP leverages a stride prefetcher to trigger loads ahead of time. When a branch retires, a read request for a trigger load is generated. Owing to the high predictability of their address, trigger load requests future addresses in advance. These requests have sufficient prefetch distance to cover the in-flight instructions and variable memory latency. As Chapter 3 shows, this can be achieved with very small structures incurring little hardware overhead.

To evaluate the results, the GAP benchmark suite [4] and the SPEC 2006 integer benchmarks [17] are used. GAP is a collection of graph algorithm benchmarks. This is one of the highest performance benchmarks available, and graphs are known to be severely limited by branch prediction accuracy. For this thesis, an 81-Kbit LDBP is integrated to the baseline 256-Kbit IMLI predictor. Results show that LDBP fixes the top-most mispredicting branches for benchmarks with very high MPKI in this study. Compared to the baseline predictor, LDBP with IMLI decreases the branch MPKI by 12% on average across all benchmarks. Similarly, the combined predictor has an average IPC improvement of 7.14%. LDBP also eases the burden on the hardware budget of the primary predictor. When combined with a 150-Kbit IMLI predictor, the branch mispredictions come down by 5.25%, and the performance gain scales by 6.63% compared to the 256-Kbit IMLI, for 9.7% lesser hardware allocation.

The rest of the thesis is organized as follows: Chapter 2 presents a background of branch prediction and prior works on branch prediction that are related to this thesis. Chapter 3

describes the different types of load-branch chains, LDBP mechanism and architecture. Chapter 4 reports the evaluation setup methodology used for this dissertation. Benchmark analysis, architecture analysis, and results are highlighted in Chapter 5. Chapter 6 concludes the thesis and explains potential future opportunities.

Chapter 2

Background

The more you know, the more you realize you know nothing.

Socrates

2.1 Overview

This chapter presents a brief overview of why branch prediction accuracy is essential to speedup processor performance and minimize energy dissipation. This chapter also analyzes recent works related to this thesis. This related work analysis also compares different types of branch predictors and their tradeoffs.

2.2 Related Work

Pipeline stalls (or hazards) leads to a pronounced degradation in performance. Pipeline hazards can be classified into three major types: data hazards, structural hazards, and control hazards. Data hazards arise due to the dependence of operand(s) from a current instruction on other in-flight instruction(s) in the pipeline. Structural hazards are caused due to conflict in hardware resources. For example, the pipeline may have one adder ALU unit, but two ADD instructions may compete for the same resource leading to the stall of the younger instruction. Branch mispredictions or any other unexpected change in Program Counter (PC) results in control hazards.

There is widespread consensus in the computer architecture research community that branch misprediction and memory bottleneck are the most significant barriers to current OoO processors' performance. Branch instructions make up a significant proportion of the instruction stream in the current general-purpose/application-specific workloads. Branch predictors aid in CPU speedup by minimizing the number of instructions executed down the wrong branch path. A highly accurate branch prediction model is essential because all the speculative actions performed after the prediction will be flushed if it is incorrect. This degrades performance as well as incurs additional energy overhead.

Branch predictors are classified into two broad categories: static and dynamic branch predictors [35]. Static branch predictors [8, 29] make prediction decisions based on meta-data like branch types/profile and/or program structure, gathered before program execution. This profile-based prediction runs multiple iterations of the program with test input sets to get a fixed

branch outcome sequence. On the other end of the spectrum, dynamic branch predictor uses run-time behavior of the workload to make predictions [20, 39, 50, 54].

2.2.1 Early branch prediction mechanisms

Loop unrolling is a trivial approach used to reduce branch mispredictions [16]. Loop unrolling involves replicating the code fragment inside the loop's body for n number of times. Here, n denotes the number of iterations in the loop. Loop unrolling eliminates the branch instruction that checks the loop termination condition. Several factors limit the effectiveness of loop unrolling:

- Loop unrolling is favorable only when the compiler can determine the size of the loop.
- Due to code replication, loop unrolling contributes to more Instruction Cache misses.
- Registers are renamed across different iterations of the unrolled loop. Such an approach leads to saturation of registers and indirectly resulting in more structural hazards.

The number of mispredictions fixed using the loop unrolling approach is minimal, and it is insignificant compared to the performance loss incurred due to incorrect branching. The most elementary branch prediction mechanisms are the "always taken" [8] and the "always not taken" [29] predictors. In these predictors, all the branches are either predicted to be taken or not taken. These static predictors use branch profiling information to determine the most frequently taken direction.

The prediction accuracy of static predictors are skewed based on the behavior of the workload. For a workload with equally distributed branch outcomes, the prediction accuracy

will be only 50%. To mitigate the penalties caused by control flow instructions in the pipelined microprocessor design and to overcome the limitations of static predictors, early attempts used a simple one-dimensional table of 2-bit counters that are indexed with the address of a branch instruction. The value of the counter is used to perform the prediction [50].

The ensuing works on branch prediction gradually raised the bar for the prediction accuracy. Yeh and Patt came up with the two-level branch predictors [54] [55]. The two-level branch predictor consists of two primary structures: the branch history register table (HR) and the branch history pattern table (PT). The HR holds the history pattern bits for the most recent branch outcomes. The branch indexes the HR (first level) at the fetch stage, and the branch history values are used to index the PT (second level) to make the actual prediction. The two-level predictor works on the following principle: the outcome of the last n branches decides the current branch's prediction. Aliasing is a very common bottleneck in such correlation-based predictors.

McFarling [28] proposed optimizations over their work. To minimize the effect of aliasing, the branch address and global-history bits are hashed. This randomizes the index used by different branches. These works leverage the high correlation between the outcome of the current branch and the history of previous branch outcomes. The skewed branch predictor [31] also aims to resolve the aliasing problem. This work uses the analogy that branch aliasing is similar to cache misses and classifies them into three types: conflict, capacity, and compulsory. Due to additional storage overhead, the skewed branch predictor does not use tags to mitigate aliasing - instead, it uses multiple branch predictor banks. These banks are indexed using different hashing functions, generated using the same metadata (branch address and global history). The banks are accessed in parallel, and the final prediction is chosen based on a majority vote scheme.

The bi-mode predictor [25] was designed to eliminate destructive aliasing in global branch history-based indexing schemes. This approach splits a traditional counter-based branch predictor into two separate direction predictors: a taken part and a not-taken part. The choice predictor picks the final prediction from one of the direction predictors. The branch history pattern is used to index the direction predictors, and the actual branch address indexes the choice predictor.

2.2.2 Neural-based branch prediction

Neural-based (perceptron) predictors have superior prediction accuracy than FSM-based predictors like bimodal predictor or *gshare* [28] due to their ability to capture more branch correlation data optimally. The cost of tracking correlation data scales linearly for a perceptron-based predictor, whereas it scales exponentially for FSM-based prediction mechanisms.

Jimenez proposed the perceptron-based branch predictor [20]. The predictor uses a simple neural network-based single-layer perceptron to represent every branch. The branch history is fed as input to the perceptrons, and the perceptrons store a vector of weights. The dot-product of the weights and input provide the output prediction. The weights are incremented or decremented by correlating the predicted outcome with the actual branch outcome. Perceptron-based predictor has low prediction accuracy for linearly inseparable functions. High prediction latency is another major drawback of this predictor.

2.2.3 History-based branch prediction

PPM-like [30] and TAGE [39] achieve higher prediction accuracy by tracking longer histories. TAGE predictor consists of a simple base predictor and a set of partially tagged predictors. Each partially tagged predictor element has a table each indexed with the global history register value of a geometrically increasing length. The tag comparison happens in all tables simultaneously. As a result, several components can yield a matching entry. The priority is given to the table that is indexed with the longest history. In case of a mismatch in all tables, the prediction calculated by the base predictor is used. Entries on the TAGE tables have a saturation counter to make the prediction and an useful counter (u). The useful counter is increased or decreased based on the correctness of the prediction. This counter also acts as an age counter, and it is periodically reset.

TAGE-based predictors are the state-of-the-art predictors, and they offer very high prediction accuracy. TAGE-based predictors fail to capture the outcome correlation of branches having an irregular periodicity or when a branch outcome history is too long or too random to capture.

2.2.4 Techniques to resolve hard-to-predict branches

Statistical correlator [40] and IMLI [47] components are augmented to TAGE to mitigate some of the mispredictions. Several studies and extensive workload analysis have identified different types of hard-to-predict branches and ways to resolve them. Sherwood et al. [49] and Morris et al. [32] proposed prediction mechanisms to tackle loop-termination branches.

The Loop Termination Buffer (LTB) [49] recognized local history-based predictors could not predict loop termination branch outcomes if the number of iterations in the loop is greater than history size. These branches can be predicted if there is a unique branch history pattern before loop exit or if the global history size is more than the loop size. The LTB has the following counters: (1) the $trip$ counter to track the number of consecutive times the loop-branch was taken before encountering a not-taken outcome; (2) a speculative and a non-speculative iteration ($iter$) counter to track the number of successive occurrences of the taken outcome. If $iter$ equals $trip$ and if the LTB is confident, then loop termination is predicted. The LTB can achieve near-perfect prediction after an initial learning period. This prediction mechanism is not effective against branches with periodically changing loop counts. The Wormhole predictor [1] improved on earlier loop-based predictors to handle branches enclosed within nested loop and branches exhibiting correlation across different iterations of the outer loop.

The Branch Misprediction Predictor (BMP) [38] is a complementary predictor. It aims only to resolve branches that are hard-to-predict by the default branch predictor. All the branches do not access the BMP, contributing to BMP's high area and energy efficiency. For a hard-to-predict branch, the BMP keeps track of the distance (number of committed branches) between two successive mispredictions. The distance correlates to the instance of the next misprediction. The BMP fixes mispredictions due to early loop termination and loop branches with variable loop size.

Several branching scenarios are better predicted when correlated with information other than traditional branch outcome history. Branches dependent on random data from load instructions contribute to a high percentage of mispredictions with TAGE-bases predictors. It is

impossible to capture the history of such branches competently, even with an unusually large predictor.

Prior works [7, 15] show that using data values as an input to the branch predictor improves the misprediction rate. The Rare Event Predictor (REP) [15] makes predictions by correlating the branch outcome against histories generated from data values. Each entry on the REP has a replacement counter (confidence counter), a prediction saturation counter, and tag fields. The replacement counter performs multiple functions. First, it acts as a confidence counter - it is incremented or decremented depending on the correctness of REP prediction. Second, it provides an effective replacement mechanism - the entry with the lowest confidence is replaced. This approach is limited by history size and if the data is highly random, it is impossible to capture branch history.

Farooq et al. [9] note that some hard-to-predict data-dependent branches manifest a specific pattern of a store-load-branch chain. They leverage this observation to mark the stores that are in the chain at compile-time and compute branch conditions based on the values of marked stores at run-time in hardware. LDBP tackles a similar problem, but the work in this theis is based on the observation that a considerable proportion of hard-to-predict data-dependent branches are dependent on the loads whose address is very predictable. Moreover, LDBP does not require any modifications to the ISA.

Gao et al. [11] proposed a closely related work. They propose the Address Branch Correlation (ABC) Predictor. They noticed that history-based predictors have low prediction accuracy for branches dependent on long-latency cache misses if the loaded data are random. The ABC predictor correlates the branch outcome to the load address and provides a prediction

based on the confidence of the correlation. Nevertheless, LDBP's approach differs in that the branch outcomes are precalculated by triggering loads that are part of the branch's dependence chain and have a highly predictable address.

3D-overrider [13] is a concurrent work that aims to resolve data-dependent hard-to-predict branches. This work only resolves load-branch chains which have one load and one simple ALU operation. They use a complex EVES [45] based load address predictor to prefetch feeder load values. The 3D-overrider uses Memory Dependence Prediction and Load Prioritization to further enhance its effectiveness. LDBP uses a simple stride-based load address predictor and resolves load-branch chains having at most 5 loads and 5 ALU operations.

Chapter 3

Load Driven Branch Predictor (LDBP)

We should be taught not to wait for inspiration to start a thing. Action always generates inspiration. Inspiration seldom generates action.

Frank Tibolt

3.1 Overview

This chapter details the load-branch chain and its types. The LDBP architecture and the tables/modules associated with it are discussed at length. I also describe the LDBP flow - how LDBP detects load-branch chains, what happens at the 'fetch' stage and 'retirement' stage of the pipeline, how loads are triggered, and many more intricate details. This chapter concludes by explaining how LDBP architecture mitigates the ill-effects of Spectre.

3.2 Load-Branch Chains

The core principle of LDBP involves exploiting the dependency between load(s) and a branch in a load-branch chain. In this sub-section, the dissertation will explain load-branch chains in detail. LDBP needs to capture the backward slice [33] of operation sequence starting from the branch. This slice's exit point must be a load with a predictable address or a trivially computable operation like a load immediate operation. There can be more than one exit point for the slice, i.e., a branch in a load-branch chain can depend on more than one parent load.

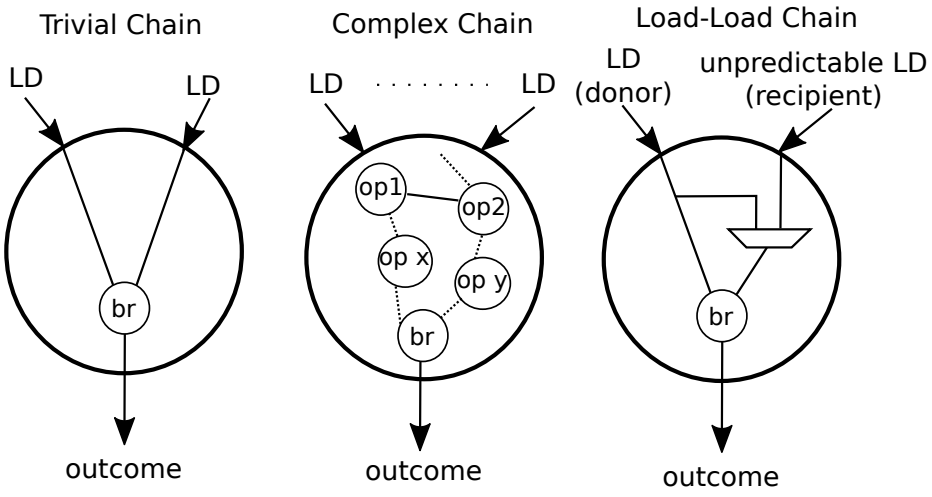

Figure 3.1: Generic load-branch chain starts with predictable loads and terminates with a branch.

As shown in Figure 3.1, the load-branch chains are classified into three different types: trivial, complex, and load-load chain. In a trivial chain, the branch has a single source operand (like a *bnez* instruction) or two source operands, and it has a direct dependency with the predictable load(s). No intermediate instructions modify the load data in this chain. For such a chain, the load-dependent branch instruction immediately follows the load(s) in program

execution order.

In a complex chain, all the branch inputs terminate with a predictable load or a load immediate. A complex chain includes at least one predictable load, one or more simple arithmetic operations, and it concludes with the branch. The LDBP framework does not track complex ALU/floating-point operations, and any chain with such an operation is invalidated. Chapter 5 describes the methodology used to determine the ideal number of loads and simple ALUs permissible in a load-branch chain. If the number of loads/ALUs exceeds this threshold, LDBP does not track that chain.

The third type of chain is a load-load chain. At first glance, it might look like the branch instruction's source operands are dependent on two loads. On deeper introspection, we notice that the data of the donor load determine the load address of the recipient load. If the address of the donor load instruction is predictable by the stride predictor, its data can be used to precompute the recipient load's address and prefetch it. Similar to the backward slice computation of the load-branch chain, a backward chain needs to be built, starting from the recipient load and ending with the donor. If the load-load chain is predictable, then LDBP can build the load-branch slice and generate predictions. Load-load chains are very uncommon in the benchmarks analyzed for this study. This implementation is a part of immediate future work.

A load-branch chain has two main constraints: (1) the maximum number of operations between the load and the branch, (2) the maximum number of input loads. For example, a chain can have five simple ALU operations before the branch. It means that a Finite State Machine (FSM) of the chain needs six cycles to compute the branch result. The number of loads triggered is directly proportional to the number of loads in a chain. Tracking a chain with a large number of

loads will saturate memory bandwidth (high trigger load demand eats up the bandwidth needed by demand requests). The thesis's benchmark analysis showed that a considerable proportion of hard-to-predict branches are part of a trivial load-branch chain.

3.3 LDBP Architecture

In this sub-section, the dissertation explains the LDBP architecture. As LDBP works in conjunction with the primary branch predictor, its architecture aims at being simple, timely, spectre-safe, and having low power overhead. The LDBP architecture is dissected into two sub-blocks: one block attached to the core's retirement stage and another block at the fetch stage. From an abstract level, the retirement block detects potential load-branch chains, creates backward slices from the branch to its dependent load(s), and generates trigger loads. On the other hand, the fetch block uses the backward slices to build FSMs of the program sequence and computes the outcome of load-dependent branches using the executed trigger load data.

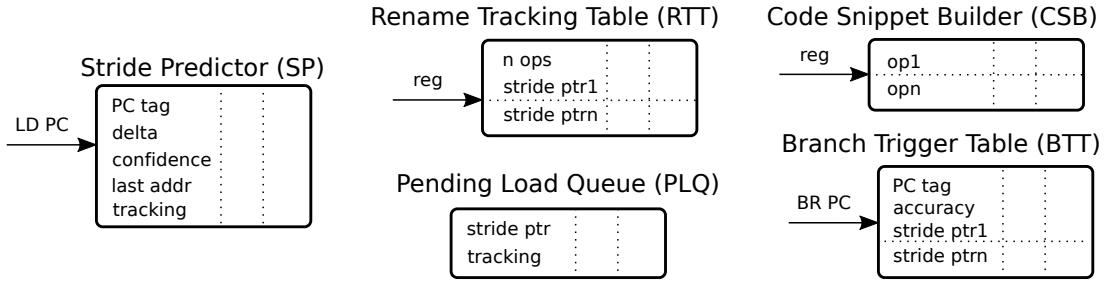

Figure 3.2: LDBP Retirement Block - Fields in each index of the tables are marked in the figure.

3.3.1 LDBP Retirement Block

A naive LDBP retirement block could consume significant power detecting and building backward slices all the time. To avoid this substantial power overhead, the thesis leverages the stride address predictor that exists in many modern microarchitectures to detect predictable loads. In addition to that, LDBP attempts to identify a load-branch chain only when these 2 conditions are true: (1) the retiring branch has low confidence with the default predictor(in this thesis, it is IMLI); (2) the load associated with the branch is predictable. Figure 3.2 shows the tables/structures associated with the retirement block.

3.3.1.1 Stride Predictor (SP)

The retiring load PC index the Stride Predictor table. This table has five fields. They are the PC tag (*sp.pctag*), the address of the last retired load (*sp.lastaddr*) [1], the load address delta (*sp.delta*), a delta confidence counter (*sp.confidence*) and a tracking bit to indicate if a given load PC is tracked as a part of a load-branch chain (*sp.tracking*).

The updating policy of the confidence counter varies across different stride predictors. Standard practice involves increasing the counter each time the delta repeats and decreasing it each time the delta changes. This approach may skew the confidence either way. Ideally, increasing the counter by one and reducing it by a higher value minimizes the bias. A tracked load (with the *sp.tracking* set) can trigger only when its confidence counter is saturated.

[1] Stride predictor can store partial load addresses to save space

3.3.1.2 Rename Tracking Table (RTT)

The Rename Tracking Table detects and builds dependencies in the load-branch chains. The retiring instruction's logical register indexes the RTT. Each table entry has a saturating counter to track the number of operations (*rtt.nops*) in a load-branch chain and a pointer list to track Stride Predictor entries (*rtt.strideptr*). The number of entries in the pointer list depends on the number of loads supported by LDBP. If a chain consists of 2 loads and 4 arithmetic operations before the branch, 3 bits are needed to track these six operations and two entries on the pointer list.

3.3.1.3 Branch Trigger Table (BTT)

The Branch Trigger Table links a branch with its associated loads and intermediate operations. The retiring branch PC indexes the BTT. Each entry has the following fields: the branch PC tag (*btt.pctag*), the list of associated loads (copied from the Stride Predictor pointer list from the RTT table (*btt.strideptr*)), and a 3-bit accuracy counter to track LDBP's accuracy for this branch (*btt.accuracy*). If the accuracy counter reaches zero, the BTT entry gets cleared, and *sp.tracking* bits of the loads in *btt.strideptr* are reset. A BTT entry is allocated only when a load-branch chain satisfies the following three conditions: (1) all loads in the chain are predictable; (2) the retiring branch has low confidence with IMLI; (3) number of loads and number of operations in the chain is within the permissible threshold.

3.3.1.4 Code Snippet Builder (CSB)

The CSB tracks the operation sequence of a load-branch chain for each logical register. Each entry on this table is a list of operations (*csb.ops*). The CSB entry is updated only when a new BTT entry gets allocated. This prerequisite ensures that the CSB is not polluted and minimizes power overhead. There are several works in the academic literature about building backward slices [33]. A table indexed by the retiring logical register (similar in behavior to an RTT) is used. It copies the chain of operations starting from the load and terminating with the branch. Initially, the possibility of combining the CSB with the RTT was considered but the idea was dropped considering the additional power dissipation this would incur. The CSB entries are only needed when a new BTT entry is populated (when a load-branch chain is established), and it would not make sense to integrate it with the RTT.

3.3.1.5 Pending Load Queue (PLQ)

The tables/structures mentioned above is sufficient to detect and build load branch chains. The PLQ acts as a buffer and stores the Stride Predictor pointer list (*plq.strideptr*) associated with a load-branch chain. It tracks whether the last retired load had a change in delta (*plq.tracking*). If there is a change, it notifies the retire block to stop triggering potentially incorrect loads. Generally, loads generate prefetches when it retires. But, in the setup used for this dissertation, the trigger load generation is delayed until the branch retires to ensure correctness in trigger load generation. The PLQ ensures that the BTT gets notified about any change in the retiring load's delta before it triggers any loads. As shown in Figure 3.4, PLQ allocates entries during BTT allocation.

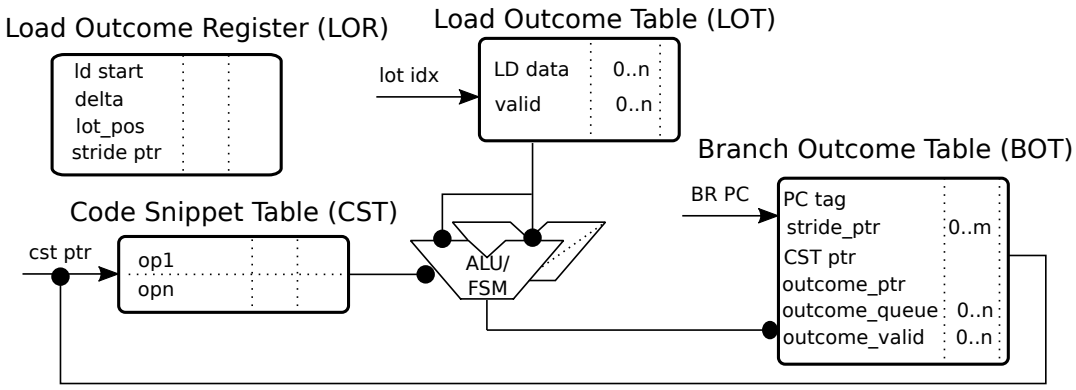

Figure 3.3: LDBP Fetch Block - Fields in each index of the tables are marked in the figure.

3.3.2 LDBP Fetch Block

The LDBP fetch block is responsible for accumulating trigger load results and computing the branch outcomes. Figure 3.3 shows the tables used by the fetch block, the registers associated with tracking loads, and the ALU used to compute the branch outcome for load-branch chains.

3.3.2.1 Load Outcome Table (LOT) and Load Outcome Registers (LOR)

The combination of LOR and LOT stores trigger load data, which could be consumed by future branches. The LOR Delta (*lor.delta*) field tracks the load address delta of each load tracked by LOR stride pointer list (*lor.strideptr*). The *lor.lot_pos* field marks the data to be used by the current branch, and it helps to queue incoming data in an appropriate LOT index. The LOT valid bit (*lot.valid*) gets set when the trigger load associated with that entry finishes execution.

The LOR keeps track of a range of load addresses whose data could be potentially useful for the current and future branches. The LOR Load Start Address (*lor.ldstart*) is the

starting load address of the range. The LOT caches the data associated with the addresses tracked by LOR. Each LOR entry has an associated LOT entry. Each LOT entry has an *n*-entry load data queue (*lot.ld_data*) and valid bit queue (*lot.valid*). The ending address tracked by LOR is $lor.ldstart + n * lor.delta$. Any trigger load address outside the address range is deemed useless, and the LOT does not cache its data.

3.3.2.2 Branch Outcome Table (BOT)

The branch PC indexes the BOT at fetch (*bot.pctag*). As shown in Figure 3.4, the BOT has two main tasks. One, use the pre-computed branch outcome to predict at the fetch stage. Two, initiate the Code Snippet Table to compute the outcome for future branches.

Each entry on the BOT has a queue of 1-bit entries holding the branch outcome (*bot.outcome_queue*). The length of this queue is equivalent to the number of entries in the LOT's load data queue (*lot.ld_data*). The *bot.outcome_ptr* points to the current BOT outcome queue entry to be used by the incoming branch instruction. BOT uses the outcome if the corresponding valid bit (*bot.valid*) is set. The BOT stride pointer list (*bot.strideptr*) has the list of loads associated with the branch. The Code Snippet FSM uses this field to pick appropriate load(s) from the LOR/LOT and the CST pointer (*bot.cstptr*) to compute the branch outcome.

3.3.2.3 Code Snippet Table (CST)

The Code Snippet Table (CST) is responsible for executing the branch backward slice to compute the branch outcome. A CST entry is allocated during BOT allocation. The CST feeds the FSMs with the operation sequence of the load-branch chain. When all the trigger load data

associated with the trigger branch are available, the FSM executes the code snippet to completion at the rate of one ALU operation per cycle. When large backward slices are supported, more FSMs are needed to reduce contention. The contention happens when all the FSM are busy. In this case, the branch outcome gets delayed until an FSM is free. As the BOT only tracks a small number of load-dependent branches, a similar-sized CST is sufficient.

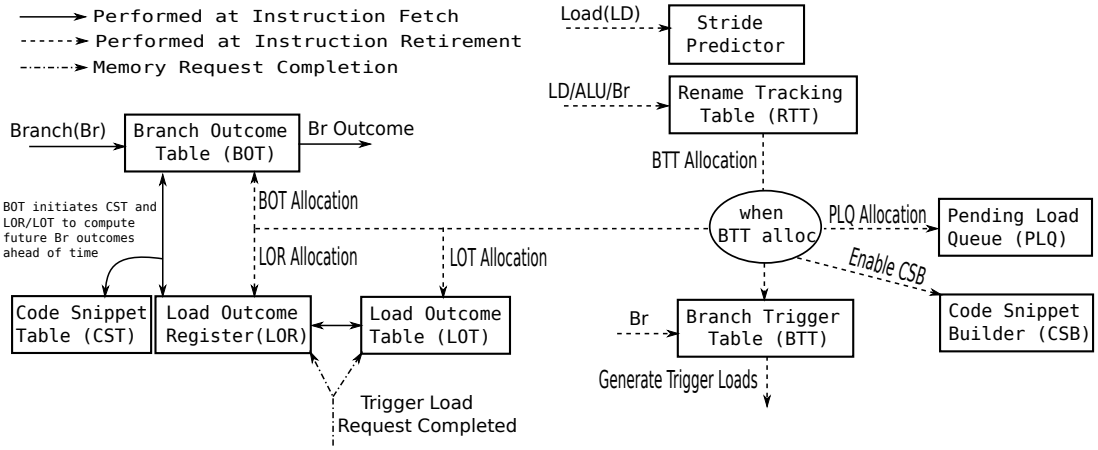

Figure 3.4: LDBP Flow - Interaction between Fetch and Retire Block.

3.4 LDBP Flow

Figure 3.4 shows the interaction between different LDBP components at instruction fetch and retirement stage. Through the rest of this section, I will explain in detail about how LDBP works.

3.4.1 Load Retirement

When a load retires, it updates the Stride Predictor. The *sp.confidence* field is updated depending upon the load address behavior. The *sp.tracking* for a load gets set at BTT allocation.

A BTT entry allocation implies that a valid load-branch chain is present, and it is necessary to track the loads in this chain to ensure LDBP triggers load(s) associated with the most active chain.

If the *sp.tracking* is set, the corresponding Stride Predictor index is appended to the Pending Load Queue table (*plq.strideptr*). The *plq.tracking* bit remains set until there is a change in delta for the load it tracks.

The retiring load also resets the RTT entry indexed by its destination register. If the *sp.confidence* is high, the *rtt.nops* is initialized to zero, and the load's pointer from the Stride Predictor is appended to the *rtt.strideptr*. In case the *sp.confidence* is low, the *rtt.nops* is saturated, and the RTT stride pointer list is cleared.

3.4.2 ALU Retirement

A retiring simple ALU operation (like addition) updates the RTT entries pointed by its destination register. The RTT retrieves *rtt.nops* and *rtt.strideptr* values pointed by its source registers [2] and accumulates it into the fields indexed by the destination register. The cumulative *rtt.nops* is represented by Equation 3.1. It is realistically infeasible to track an infinitely large load-branch chain. So, there is a threshold on the number of operations and the number of loads supported by LDBP. If $rtt[dst].nops$ exceeds the limit, the corresponding RTT entry gets invalidated. For simplicity, the number of operands per source are added, ignoring any potential redundancy in operations.

[2] At most two sources in RISC-V

$$rtt[dst].nops = rtt[src1].nops + rtt[src2].nops + 1 \tag{3.1}$$

LDBP does not support complex operations like multiplication or floating-point operations. As a result, when one of these instructions retire, the RTT entry indexed by it is invalidated to ensure a load-branch chain does not get polluted by complex operations.

3.4.3 Branch Retirement

At cold start, when a branch retires, it indexes the RTT only when it has low confidence with the default IMLI predictor [3]. BTT entry gets allocated only when all the loads in this chain are predictable and $rtt[dst].nops$ is below permissible threshold.

3.4.3.1 BTT Allocation

On BTT allocation, the contents of the *rtt.strideptr* are copied to the $btt.strideptr$. The *btt.accuracy* counter is initialized to half of its saturation value. The *sp.tracking* bit for the associated loads are set, and the CSB starts building the code snippet for this load-branch chain. As shown in Figure 3.4, the BTT allocation creates a chain reaction by initiating the PLQ allocation, LOR/LOT allocation, and BOT allocation.

Each load associated with the branch has a unique entry during LOR/LOT allocation. Load-associated metadata from the Stride Predictor populates the LOR fields. The *lor.lot_pos* is cleared. Similarly, BOT entry gets reset on allocation, and *btt.strideptr* updates the stride pointer list on the BOT. The branch's PC tag is assigned to *bot.pctag*.

[3] IMLI is confident when the longest table hit counter is saturated.

3.4.3.2 BTT Hit

On BTT hit, the *btt.accuracy* counter gets incremented if LDBP made a correct prediction, and the default IMLI predictor mispredicts and vice versa. If this counter reaches zero, the BTT deallocates the entry and the *sp.tracking* associated with *btt.strideptr* are cleared.

The CSB starts to build the code snippet for the load-branch chain on BOT allocation. After CSB completes the snippet, on a BTT hit, the code snippet is copied to the CST. The CSB is disabled after this process.

When the retiring branch hits on the BTT, it reads the corresponding PLQ entries to ensure if the tracking bit is high for the loads in the *btt.strideptr*. The BTT can trigger load(s) if the PLQ and LOR track all the associated loads. Equation 3.2 represents the address of the load triggered. The *lor.ldstart* is incremented by load address delta to ensure better coverage after every trigger load generation. The *lor.lot_pos* is incremented when a new load is triggered. The trigger load distance (*tl_dist*) and the number of triggers generated for each load can be tuned to facilitate better load timeliness.

$$tl\_addr = lor.ldstart + lor.delta * tl\_dist \tag{3.2}$$

3.4.3.3 Recovery on load delta change or change in chain path

There can be scenarios where the load-branch chain might change. It could happen when a different operation sequence is taken to reach the branch. There are situations where the delta associated with any of the branch's dependent loads might change, potentially resulting in triggering incorrect loads. During such occurrences, LDBP flushes the branch entries on the

BTT, BOT (and its associated CST entry), and its corresponding load entries on the LOR/LOT. The tracking bit on the Stride Predictor and PLQ are reset for these loads. Such an aggressive recovery scheme guarantees higher LDBP accuracy and reduced memory congestion due to unwanted trigger loads.

3.4.3.4 Trigger Load Completion

When a trigger load completes execution, it checks for matching entries on the LOR. There could be zero or more entries on the LOR, which could have the address range of this completed request. The address is a match on the LOR entry if it is within the LOR entry's address range and is a factor of the *lor.delta*. On a hit, the corresponding LOT entry stores the trigger load data in the *lot.ld_data* queue, and its valid bit is set. The LOT data queue index is computed using Equation 3.3a and 3.3b.

$$lot\_id = \frac{(tl.addr - lor.ldstart)}{lor.delta} \tag{3.3a}$$

$$lot\_index = (lor.lot\_pos + lot\_id)\%lot.ld\_data.size() \tag{3.3b}$$

3.4.4 Branch at Fetch

When a branch hits on the BOT at instruction fetch, the *bot.outcome_ptr* is increased by one. This is the only value speculatively updated in the LDBP fetch block. When there is a table flush due to misprediction, load-branch chain change or load delta variation, the *bot.outcome_ptr* gets flushed to zero. The BOT outcome queue entry pointed by the *bot.outcome_ptr* yields the

branch's prediction.

The *bot.cst_ptr* proactively instigates the computation of future branch outcomes at fetch. The CST FSMs use the load data values from valid entries on the LOT. Once the outcome is computed, the corresponding *bot.outcome_queue* entry gets updated.

3.5 Spectre-safe LDBP

Timing side channel leaks are a powerful tool for hackers. In the late 90s [23], it became known that many encryption algorithms were susceptible to time side channel attacks. Specifically, it was observed that different data had different branch prediction performance, and this information leak could be observed by other applications or code sections, which compromised the algorithms. More recently, the Spectre [22] class of attacks leverage time changes in the cache caused by speculatively executed instruction. In all these attacks, the information is leaked because the attacker has a clock or performance counter from the processor, and is able to measure the time impact resulting from executing some code. All these leaks could be avoided if the attacker did not have the capacity to gather any performance information on the code under attack.

LDBP architecture is carefully designed to ensure that there is no information leakage through the branch predictor. The LDBP has been designed to avoid speculative updates. The reason is not to create another source of Spectre-like [22] attacks. The LDBP retirement block is only updated when the instructions are not speculative. This means that it never has any speculative information and potential speculative side-channel leak.

The LDBP fetch block is populated only with information from the retirement block. Even the trigger loads are sent when a safe target branch retires. The only speculatively updated field is the LOR table, but this table is flushed after each miss prediction or load delta change, and the state is rebuilt from the LDBP retirement block.

In a way, the LDBP is not a new source of speculative leaks because it is only updated with safe information, and the fields updated speculatively are always flushed on any pipeline flush. The flush is necessary for performance, not only for Spectre. The reason is that when the ”number of in-flight” trigger loads change due to flushes, the LOR must be updated. LDBP structures are not source of speculative leaks, but the loads in the speculative path can still leak unless speculative loads are protected like in [37]. The result is that LDBP is not a new source of speculative leaks like most branch predictors that gets speculatively updated and not fixed on pipeline flushes.

3.6 Multiple Paths Per Branch

The LDBP load-branch slices are generated at run-time, and they can cross branches. As a result, the same branch can have multiple chains or backward slices. These cases are sporadic in benchmarks from GAP as they have a large and somewhat regular pattern. Multi-path branches are slightly more common in the SPEC CINT2006 benchmarks.

The analysis performed as a part of this work shows that branches with multiple slices are not frequent, and when they happen, they tend to depend on unpredictable loads. Therefore, it is not a significant cause of concern for LDBP in these cases. Nevertheless, it can be an issue in

other workloads. The dissertation leaves it as a part of future work and possibly find benchmarks that exhibit such behavior more predominantly.

Chapter 4

Setup

A complex system that works is invariably found to have evolved from a simple system that works.

John Gaule

4.1 Overview

In this chapter, the thesis briefly talks about the infrastructure used to run simulations to evaluate the effectiveness of LDBP. This chapter also provides details about the benchmarks used, baseline processor configuration, and models against which LDBP was compared.

Parameter	Size
L1 Instruction Cache	32 KB, 8-way
L1 Data Cache	32 KB, 8-way
L2 Cache	512 KB, 8-way
Laod Queue	44 entries
Store Queue	40 entries
Reorder Buffer (ROB)	224 entries
Physical Register File	180 entries
Integer Execute Unit	4
Integer Execute Unit Scheduler	16 entries
Address Generation Unit	3
Address Generation Unit Scheduler	28 entries
Floating-point Execute Unit	4
Issue Width	6
Retire Width	8

Table 4.1: Some architectural parameters for AMD Zen-2 core.

4.2 Simulation Setup

In this thesis, ESESC [2] is used as the timing simulator. ESESC is a cycle-accurate multi-core simulator that can model in-order as well as out-of-order cores. It is an execution-driven simulation model. ESESC supports RISC-V and MIPS Instruction Set Architecture. ESESC models bandwidth and contention for core and memory hierarchy. Industry (MIPS) has correlated ESESC with RTL with less than 10% error for SPEC-like benchmarks. The processor configuration is set to closely model an AMD Zen 2-like core [51]. Some of the parameters of the Zen 2 core are given in Table 4.1.

Table 4.2 shows the Instructions Per Cycle (IPC) and MPKI for the benchmarks investigated when running the baseline 256-Kbit IMLI predictor. To match the Zen 2 architecture, the baseline branch prediction unit has a fast (1 cycle) branch predictor and a slower but more accurate (2 cycle) IMLI branch predictor. The baseline configuration is evaluated against 1-Mbit

Benchmark	Branch MPKI	IPC
spec06_xalan	1.0	1.97
spec06_sjeng	5.8	1.78
spec06_perlbench	1.8	1.42
spec06_omnetpp	1.5	2.73
spec06_mcf	6.2	1.21
spec06_libquantum	0.0	1.17
spec06_hmmer	12.9	2.42
spec06_h264ref	0.9	3.01
spec06_gobmk	12.2	1.55
spec06_gcc	0.3	2.15
spec06_bzip2	2.9	2.75
spec06_astar	14.9	0.91
gap_tc	44.4	1.07
gap_sssp	6.2	0.89
gap_pr	4.6	1.65
gap_cc	32.7	0.51
gap_bfs	23.7	0.65
gap_bc	22.0	1.17

Table 4.2: Benchmarks used and their MPKI and IPC running baseline 256-Kbit IMLI.

IMLI, and different IMLI configurations (150-Kbit, 256-Kbit, and 1-Mbit) are augmented with an 81-Kbit LDBP.

4.3 Application Setup

In this dissertation, LDBP was evaluated using SPEC 2006 integer benchmarks and the GAP Benchmark Suite (GAPBS). All the benchmarks are compiled with *gcc 9.2* with *-Ofast -flto* optimization for a RISC-V RV64 ISA [52]. For SPEC CINT2006, all the benchmarks were run, skipping 8 billion and modeling for 2 billion instructions. All the GAP applications were run with "*-g 19 -n 30*" command line input set and instrument the benchmarks to skip the initialization, as suggested by GAP developers. The command line argument for GAPBS can be

interpreted as follows: run the benchmark on a graph with 2^{19} vertices (*-g 19*) for 30 iterations (*-n 30*).

As shown in Table 4.2, benchmarks in SPEC CINT 2006 have branch misprediction rates spanning across different ranges. Benchmarks like *astar* have a high misprediction rate, whereas *libquantum* has near-perfect branch prediction accuracy with the baseline Zen-2 core with IMLI predictor. The SPEC 2006 floating-point benchmarks have pretty solid branch prediction accuracy. So, I felt including it for evaluation will not be a true reflection of this dissertation's purpose.

In contrast, the GAP benchmarks have very high MPKI - one of the main reasons for picking this benchmark suite. The GAP benchmark suite is a pack of high-performance implementations written with C++11. The input datasets of GAPBS includes a combination of real graphs and synthetic graphs [3].

Chapter 5

Results and Analysis

> One can measure the importance of a scientific work by the number of earlier publications rendered superfluous by it.
>
> David Hilbert

5.1 Overview

In this chapter, the results of the thesis are highlighted. This chapter presents the performance metrics (IPC and branch misprediction), benchmark analysis, sensitivity study for different LDBP components, energy implications of using LDBP, and the importance of trigger load timeliness.

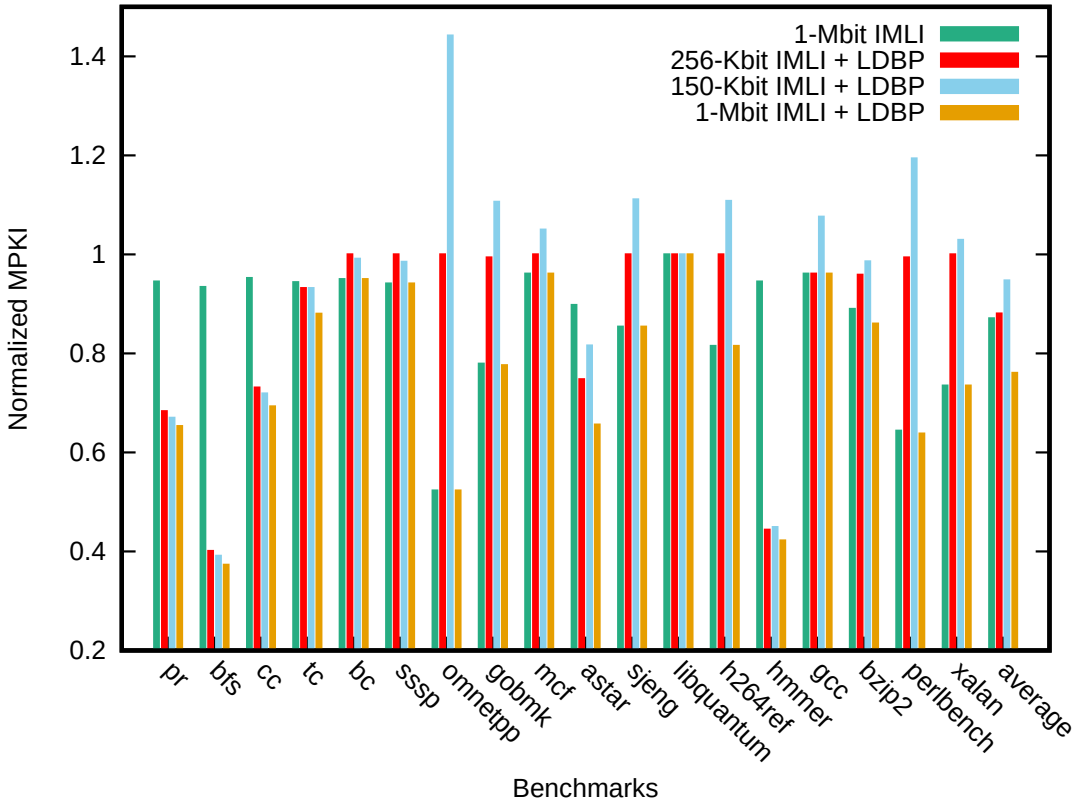

Figure 5.1: LDBP minimizes the mispredictions by more than 12% when combined with the baseline 256-Kbit IMLI.

5.2 IPC and Branch Misprediction Rate

In this section, the thesis compares the performance, and misprediction rate variations between the baseline IMLI predictor and the proposed LDBP predictor augmented to IMLI. Mispredictions Per Kilo Instruction (MPKI) is the metric used to compare the misprediction rate in this section.

Figure 5.1 shows the normalized MPKI values compared to the baseline IMLI for different branch predictor configurations. LDBP has a considerable impact on more than half of the benchmarks. On average, the IMLI 256-Kbit + LDBP predictor reduces the MPKI of

GAP and SPEC CINT2006 benchmarks by 12%. As shown in Table 4.2, SPEC CINT2006's *astar* has poor branch prediction accuracy. The most mispredicting branch in *astar* constitutes 22% of the benchmark's mispredictions. This branch has a direct dependency with a load, but LDBP cannot fix this branch as the address of the load feeding this branch has a fluctuating delta. LDBP manages to minimize *astar's* total branch misses by 16.6% without fixing the most mispredicting branch. These numbers attest to the fact that a considerable proportion of hard-to-predict branches on most benchmarks depend on data from loads with a predictable address. Another observation to note is that quadrupling the size of IMLI fixes only 12.9% branch misses across all benchmarks, compared to the baseline. This inference substantiates the fact that a huge TAGE-like predictor cannot efficiently capture the history of hard-to-predict data-dependent branches.

Figure 5.2 compares IPC changes over baseline 256-Kbit IMLI for different branch predictor configurations. LDBP was able to achieve an average IPC improvement of 7.14% when paired with the baseline predictor. An interesting observation is that the GAP benchmarks have a speedup of 13.6% with this configuration. In contrast, they have a slightly better IPC gain of 13.9% over the baseline when running on 150-Kbit IMLI + LDBP. The reason for this trend is that a smaller IMLI can fix lesser branches, and LDBP fixes branches that have low confidence with IMLI. Therefore, lower the MPKI of the primary predictor, more the work for LDBP. For some benchmarks like *bfs*, the smaller predictor even outperformed its larger counterpart. Moreover, the 150-Kbit IMLI + 81-Kbit LDBP offers 6.6% higher performance gain and 5.3% lesser branch misses than the baseline 256-Kbit IMLI for a 9.7% lower hardware budget.

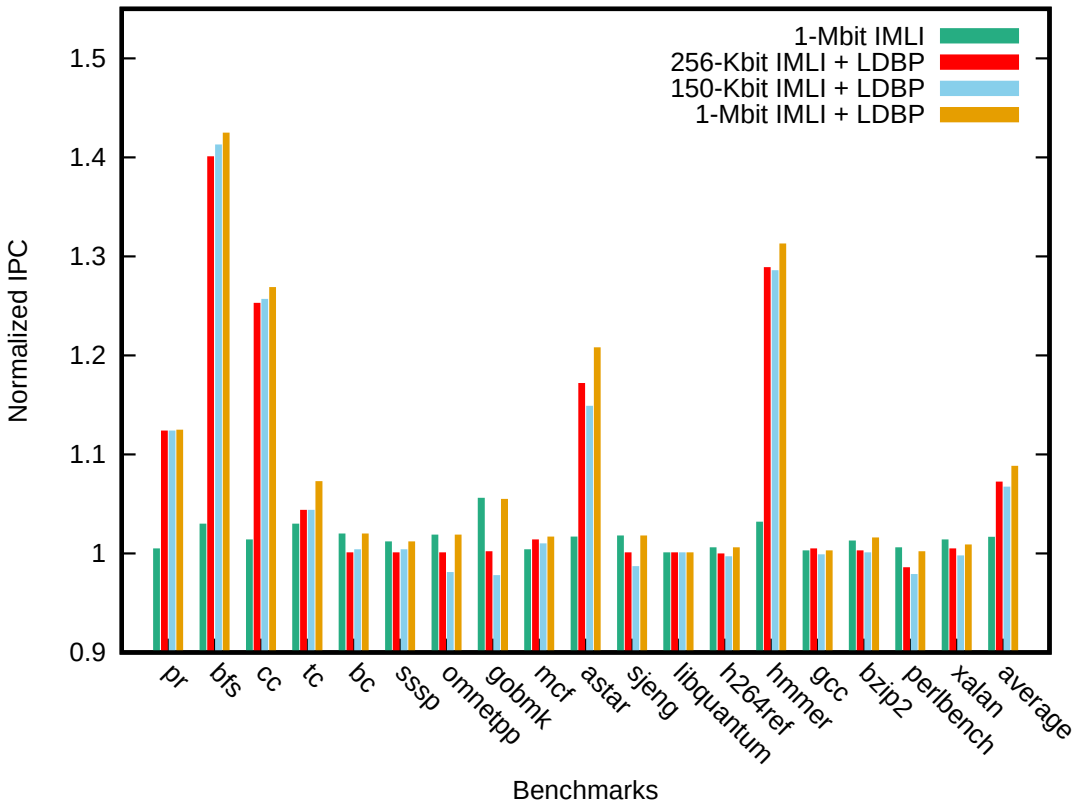

Figure 5.2: LDBP (when combined with 150-Kbit IMLI or 256-Kbit IMLI) outperforms the large 1-Mbit IMLI comprehensively.

The MPKI and performance improvements yielded by LDBP clearly shows that hard-to-predict load-dependent branches are major contributors to overall mispredictions in benchmarks across different application suites. LDBP does not affect some benchmarks like *mcf*, *sssp* and *bc*. This behavior can be attributed that mispredicting branches in these benchmarks do not have a load-branch dependency that can be captured by LDBP. An anomaly to note on Figure 5.1 and 5.2 is the behavior of *gobmk* running with 150-Kbit IMLI + LDBP. It can noticed that the IPC decreases by 2%, and the MPKI worsens by 10%. It is because the 150-Kbit IMLI has a worse MPKI and IPC compared to the baseline IMLI. Added to that, LDBP does not yield

```
1 for(NodeID u=0; u < g.num_nodes(); u++){
2   if(parent[u] < 0){
3       ..
4       ..
5       ..
6   }
7 }
```

Figure 5.3: GAP's BFS source code snippet.

any improvement for *gobmk*.

5.3 Benchmark Study

In this section, the thesis analyzes examples from different benchmarks where LDBP works and cases where LDBP doesn't work.

5.3.1 Case 1: BFS (GAP Benchmark Suite)

For the first case study, the Breadth-First Search (BFS) algorithm is explored. It is one of the most popular graph traversal algorithms used across several domains. Figure 5.4 and Figure 5.3 shows a snippet of RISC-V assembly and its corresponding pseudo code from GAP's BFS benchmark. Here, the loop traverses over all the nodes in the graph to assign a parent to each node. The arbitrary nature of the graph makes it hard to predict if a node has a valid parent as each node can have multiple possible edges, but the node traversal is in order. It is hard to predict $parent[u]$, but u is easily predictable (Line 2 in Figure 5.3). The branch in Line 9 in Figure 5.4 is the most mispredicted branch in this benchmark. It contributes to about 29% of all mispredictions when simulated on the baseline architecture with 256-Kbit IMLI. When LDBP is augmented into this setup, it resolves 96% of the mispredictions for this branch and reduces the

```
1 L2: addi a7,a7,1
2     ld  a5,8(t5)
3     bge a7,a5,L1 //outer 'for' loop
4     sext.w t6,a7
5     slli t1,a7,0x2
6     ld  a5,0(a1)
7     add t1,t1,a5
8     lw  a5,0(t1)
9     bgez a5,L2  //'if' condition check
```

Figure 5.4: GAP's BFS RISC-V Assembly code for Figure 5.3.

overall MPKI of BFS by 59.9%. It is also instrumental in gaining 40% speedup.

5.3.2 Case 2: HMMER (SPEC CINT 2006)

Figure 5.5 shows the RISC-V assembly code section of the branch (line 8) contributing to most misprediction in SPEC CINT2006 *hmmer*. It accounts for 39% of all mispredicted branches. The branch outcome is dependent on values from different matrices. The randomness of the data involved makes this a very hard-to-predict branch. Each branch source operand is dependent on two loads. As the benchmark traverse over matrices, the loads involved in this case has a traceable address pattern. LDBP has to track four different loads and some intermediate ALU operations to make the prediction. LDBP fixes 67.4% of the mispredictions yielded by *bge*. Appending LDBP to the baseline IMLI improves the IPC of *hmmer* by 29% and reduces the overall MPKI of this benchmark by 56%.

5.3.3 Case 3: PR (GAP Benchmark Suite)

Figure 5.6 represents the code snippet containing the two most mispredicting branches in the PR benchmark (lines 5 and 9). *beq*1 and *beq*2 contribute to about 37.2% and 62.6% of all

```
lw     s11, 0(a3)
lw     a3, 4(a7)
addw   a3, s11, a3
sw     a3, 0(t3)
lw     s10, 0(s10)
lw     s11, 4(t1)
addw   s11, s10, s11
bge    a3, s11, LABEL
```

Figure 5.5: SPEC CINT 2006 hmmer RISC-V assembly code.

```
ld     a3, 8(a2)
ld     a4, 0(a2)
..
..
beq    a3, a4, L1   /* beq1 */
..
..
addi   a4, a4, 4    /* target L2 */
..
..
beq    a3, a4, L2   /* beq2 */
```

Figure 5.6: GAP PR (PageRank) RISC-V assembly code.

misprediction in PR, respectively. LDBP reduces the MPKI of PR by 32%. The load addresses are predictable, but the stride does not remain constant for a prolonged period. LDBP fixes 84% of all *beq*1 mispredictions. No improvements were made for *beq*2. For *beq*2, when the branch is taken, the value of *a*4 is modified by *addi*. This ALU is part of a different path than the original load-branch chain for *beq*2. Though the loads are predictable, due to the multi-path nature of *beq*2, it is not effective with LDBP. As a part of future work, I plan to add architectural support in LDBP to track dual-path chains.

```
1  START:  lw      a2,0(a3)
2          addi    a3,a3,4
3          bge     a1,a2,SKIP
4          mv      s6,a4
5          mv      a1,a2
6  SKIP :  addiw   a4,a4,1
7          bne     s10,a4,START
```

Figure 5.7: SPEC CINT2006 sjeng RISC-V assembly code.

5.3.4 Case 4: SJENG (SPEC CINT 2006)

There exist some load-branch chains that cannot be detected by LDBP at the moment. Let us consider an example from SPEC CINT2006 sjeng. Figure 5.7 shows a fragment of RISC-V assembly from sjeng, which has the most mispredicted branch (line 3). The code performs a sort operation. The loop iterates to find a value greater than or equal to $a1$. If it finds such an instance, that value is assigned to $a1$ through the *mv* (move) instruction at line 5. LDBP cannot detect this pattern as one of the branch sources($a1$) is generated by the *mv* instruction, and it is independent of a load. Even if the load instruction (line 1) is predictable, LDBP cannot fix this branch. Adding support in LDBP to track these sort-based load-branch chains is part of future work.

5.4 Trigger Load Timeliness

In this section, the thesis will focus on trigger load prefetch distance and its importance in achieving optimum LDBP timeliness. I will use Figure 1.1 to highlight the criticality of timely trigger loads. This example is the vector traversal problem discussed in Section 1.1. In this example, let us assume a scenario where it takes six cycles to load data from the vector, and there

are ten in-flight load-branch iterations. As the load address has a delta of 8, to achieve an IPC of 1, the new trigger load needs to be sent at least 16 cycles ahead. If the current load address is x, LDBP triggers a load address with a distance of 16 ($x + 8 * 16$). In reality, it would be ideal to use even a larger distance to compensate for variable memory latencies.

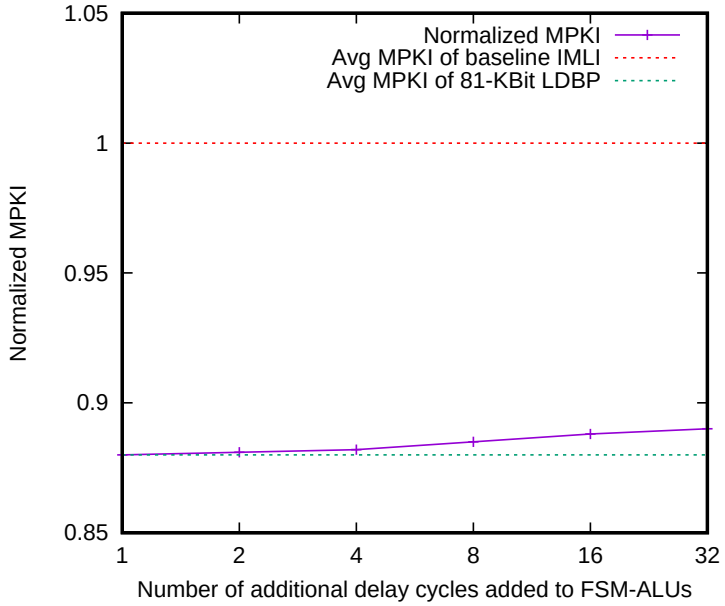

Figure 5.8: Sensitivity of LDBP when adding extra delays to FSM-ALUs.

To understand how timely the trigger load requests are in LDBP, artificial extra delays are injected. Figure 5.8 shows the variation of MPKI compared to baseline IMLI, when random extra delay cycles are added. The green dotted lines denote the MPKI of baseline IMLI+81-KBit LDBP normalized to the baseline. This clearly show that early triggering of loads acts as buffer for unexpected pipeline delays/variable memory latency. This ensure that the FSM-ALUs have sufficient time to pre-compute the branch outcome. Even for an extra delay of 32 cycles, the MPKI variation is less than 1%.

At a first look, Figure 5.8 seems counter intuitive. It says that even adding 32 cycles delay has a small impact on the overall MPKI. The reason is that LDBP is aggressively sending trigger loads. Also, any misprediction flushed the pipeline but not the triggered load. Since LDBP targets code sections with frequent mispredictions, it has extra time to cover memory latency overhead. The result is that most LDBP predictions are timely.

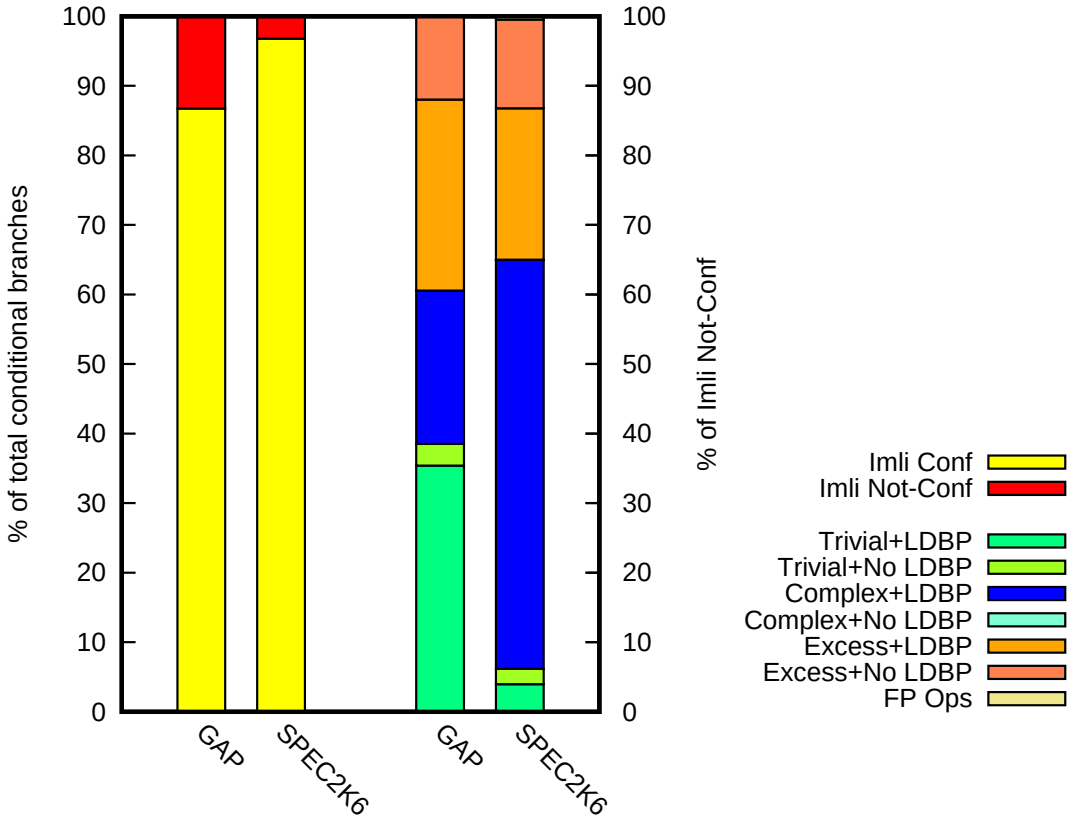

Figure 5.9: Load-branch chain classification for GAP and SPEC CINT2006 shows the proportion of IMLI mispredicted branches (IMLI no conf) that are dependent on loads with predictable addresses.

5.5 Load-Branch Classification

To understand what happens with the IMLI mispredicted branches, a limit study is performed. Each IMLI mispredicted branch is characterized based on the LDBP backward slice. If the backward slice terminates in non-predictable load addresses, the slice is referred to as "No LDBP". The backward chains can be broadly categorized into three different type: Trivial, Complex, and Excess. Trivial chain does not have any ALU operation in the backward slice. Complex chain has atmost 5 ALU operations, and Excess has more than 5 ALU operations.

Figure 5.9 shows that IMLI does a good job, and less than 5% of the branches are marked as "IMLI Not-Conf" for SPEC CINT2006. From those branches, for both SPEC CINT2006 and GAP, over 50% of the branches are LDBP trivial or complex. Figure 5.1 shows that LDBP reduced MPKI by 12%. Section 5.4 has shown that the problem is not due to timeliness. The reason is that the "LDBP" classification considers a load predictable if a stride prefetcher can predict a significant part of the loads. The issue is that the stride prefetcher is limited due to outliers or changes of deltas. If the stride prefetcher was correct more frequently a significant part of branches could be fixed. This is an area that shows potential additional opportunities for LDBP.

5.6 LDBP Sizing

This section explains the methodology used to size the tables in LDBP. The variation in MPKI for a different number of entries in each structure in the predictor is analyzed. To have a fair LDBP table sizing, a baseline LDBP is defined When the MPKI sensitivity for a table's

size is analyzed, all other tables in LDBP have infinite entries. Such an approach ensures a fair estimation of the table's impact on LDBP accuracy. A 5% MPKI increase from infinite LDBP is the cut-off used to determine the ideal table size. For this study, all the benchmarks from SPEC CINT2006 and GAP having MPKI reduction greater than 1% with the infinite LDBP are used. I did not include all benchmarks for sizing because some benchmarks had no effect on MPKI even for an infinite size LDBP. Adding them to this sizing study will not be a true reflection of the impact of LDBP.

Structure Name	**No. of Entries**	**Total Size (Kbit)**
Stride Predictor	32	1.59
Rename Tracking Table (RTT)	32	3.09
Pending Load Queue (PLQ)	32	0.22
Branch Trigger Table (BTT)	8	0.88
Code Snippet Builder (CSB)	32	5
Load Outcome Register (LOR)	16	1.44
Load Outcome Table (LOT)	16	65
Branch Outcome Table (BOT)	8	1.93
Code Snippet Table (CST)	8	2.5
Total LDBP Size		**81.65**

Table 5.1: Overall LDBP Size is 81-Kbit.

Table 5.1 shows overall size of LDBP and the breakdown of individual table sizes. The overall size for the LDBP is 81-Kbit. As a reference, the IMLI predictor used is 256-Kbit. The fetch block in a processor like a Zen 2 also includes a 32-KByte instruction cache and two-level BTBs with 512 and 7K entries. The largest LDBP table is the LOT that can use area-efficient single port SRAMs.

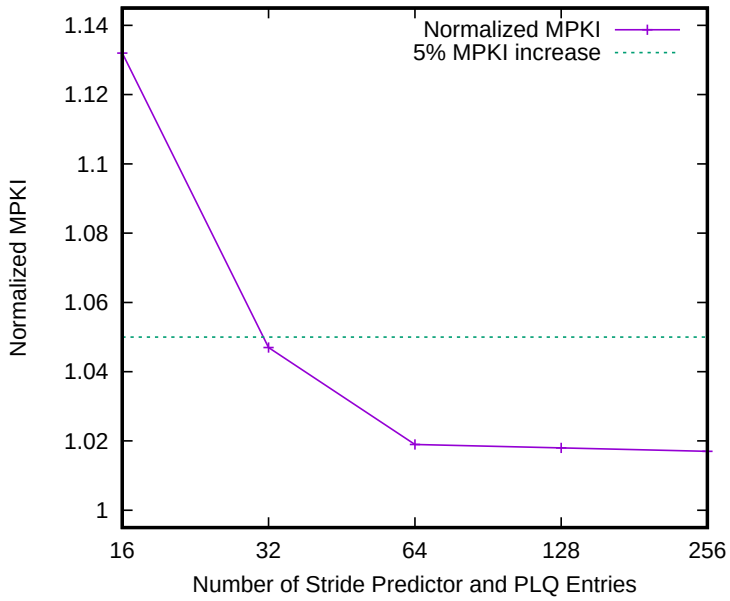

Figure 5.10: 32 entries are sufficient in the Stride Predictor and PLQ to achieve prediction accuracy varying by less than 5% from the infinite LDBP.

5.6.1 Stride Predictor and PLQ Sizing

Figure 5.10 shows the impact of the number of entries on the Stride Predictor and the PLQ on MPKI. It can observed that the MPKI drop is just under 5% when the number of entries is around 32. With reduced stride predictor and PLQ entries, a load tracked as a part of the hard-to-predict load-dependent branch's chain can be evicted to make way for a new incoming load. LDBP cannot determine if a load is trigger-worthy if it is not in the stride predictor table. Entries larger than 64 have a negligible effect on the MPKI. The stride predictor and PLQ have 32 entries each as it offers the perfect equilibrium between MPKI and hardware size. As shown in Table 5.1, the stride predictor has a storage budget of 1.59-Kbits, and the PLQ occupies 0.22-Kbits.

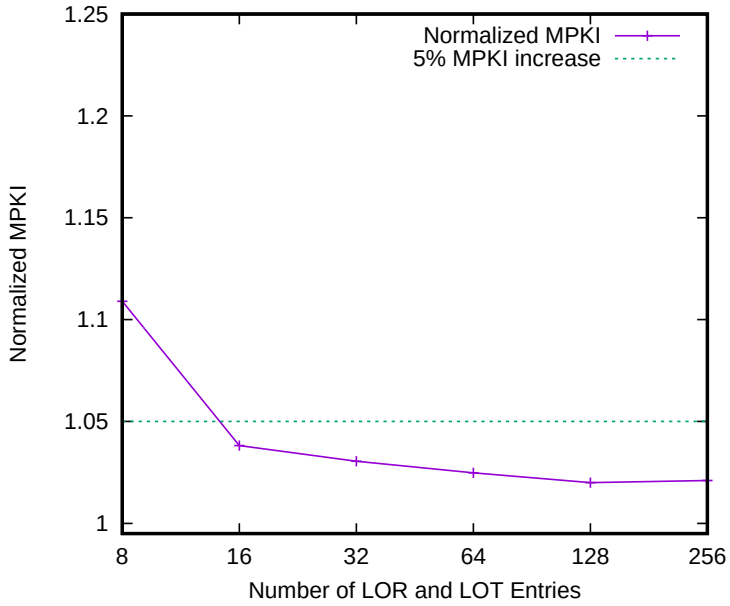

Figure 5.11: Tracking 16 loads on the LOR and LOT is adequate to maintain high LDBP accuracy.

5.6.2 LOR and LOT Sizing

Figure 5.11 plots the effect of varying LOR size on MPKI. There is a sharp increase in MPKI when the number of trigger loads tracked is less than 16. At the 5% cut-off point, LOR and LOT has around 12 entries. To minimize the impact of the sharp drop in MPKI, 16 entries are allocated to both the LOR and LOT. The necessity to store the complete load data contributes to the large size of the LOT. The number of entries on the LOT data queue is determined by how proactively LDBP wants to predict branches and trigger its associated loads. The number of entries on the BOT's outcome queue matches the LOT data queue entries. The sizing of the BOT outcome queue is discussed in Section 5.6.3.

Some load-dependent branches may consume two or more trigger loads. A bottleneck on the number of trigger loads tracked has a direct implication on the effectiveness of LDBP.

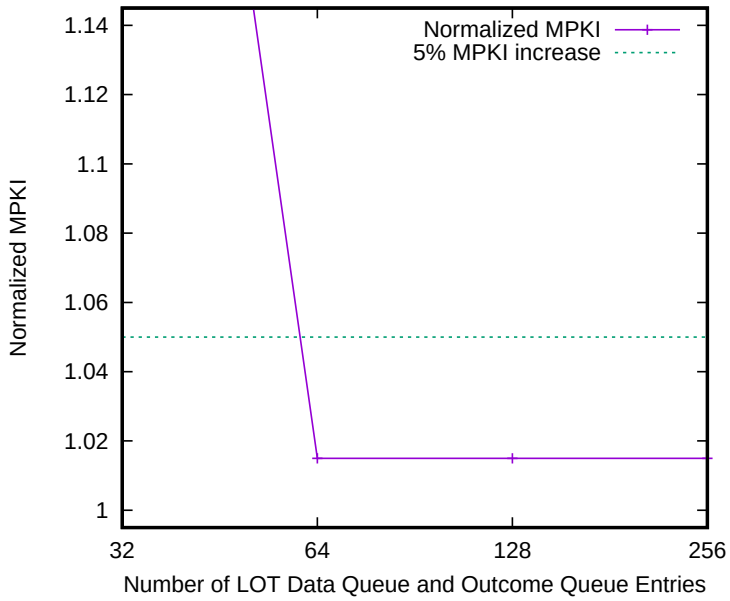

Figure 5.12: The LOT Data Queue and Outcome Queue requires 64 entries each.

In most cases, the load-dependent branch tends to be the entry-point to a huge loop. In such cases, it sufficient for LDBP to track just one branch and its associated trigger loads. Therefore, a reasonably small to medium number of entries on the LOR and LOT is adequate to maintain LDBP accuracy.

5.6.3 Outcome Queue/LOT Data Queue Sizing

The outcome queue is part of the BOT. The criticality of the outcome queue in the overall scheme of LDBP warranted optimal sizing. The number of entries in this queue correlates to the number of future outcomes trackable for a given branch PC. The outcome queue entries directly impact the number of entries on the LOT data queue. It is sufficient for the LOT data queue to have as many entries as the branches tracked by the outcome queue. From Figure 5.12, the ideal number of outcome queue entries at the cut-off point is 64. As the outcome queue

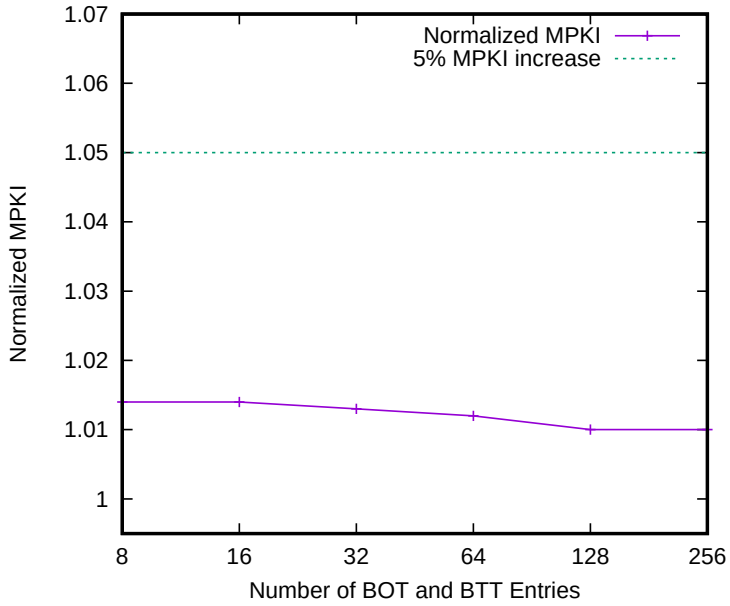

Figure 5.13: The effectiveness of LDBP remains steady for different number of entries on the BOT and BTT tables. It is sufficient to track at most 8 branches on these tables.

size decreases, the MPKI increase gets steeper. A smaller outcome queue inhibits the ability of LDBP to trigger loads with higher prefetch distance. On the flip side, the outcome queue size larger than 64 almost hits an MPKI plateau.

5.6.4 BOT and BTT Sizing

Figure 5.13 shows the variation of MPKI for different sizes of BOT and BTT. Just like the LOR and LOT, a small to a medium number of entries on the BOT and BTT is sufficient to track almost every load-dependent branch in an application. These branches are usually part of large loops. These huge loops give LDBP adequate time to capture the new branch-load chain even if they replace an already existing entry from the tables. The correlation between the number of entries and MPKI has very minimal variations. Therefore, it is sufficient to have just

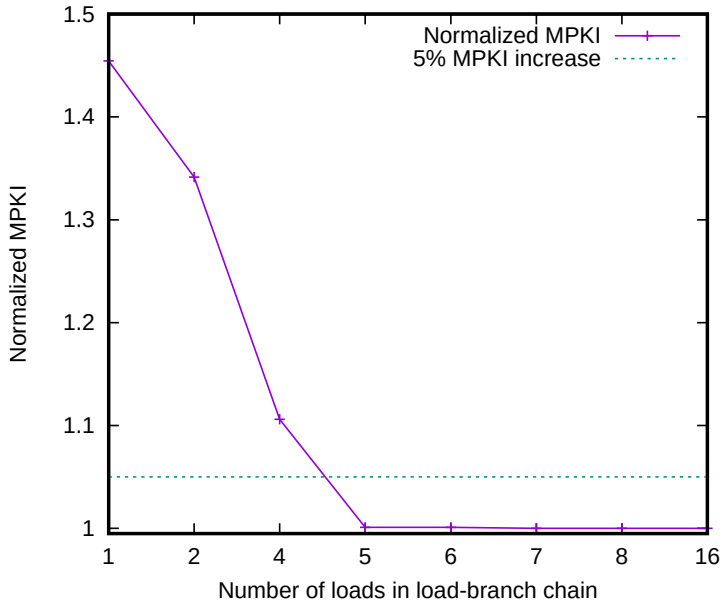

Figure 5.14: LDBP must track at least 5 loads to maintain healthy prediction accuracy.

8 entries on the BOT and BTT.

5.6.5 CSB and CST Sizing

The CSB builds the load-branch slice. It is critical to size this table optimally to keep LDBP's hardware budget under check. Figure 5.14 and 5.15 shows the change of MPKI for different load and ALU operations threshold in an LDBP chain. Five loads and four ALU operations are needed to ensure maximum LDBP efficiency. These figures reflect the cumulative number of operations tracked by both the source operands of a branch instruction. Each source operand of the branch might need to track only fewer operations.

Figure 5.16 shows the number of sub-entries needed by each CSB index. This figure clearly shows that it is sufficient for each branch source register to track five operations to support an LDBP chain with a maximum of nine operations. There are 32 entries on the CSB, and each

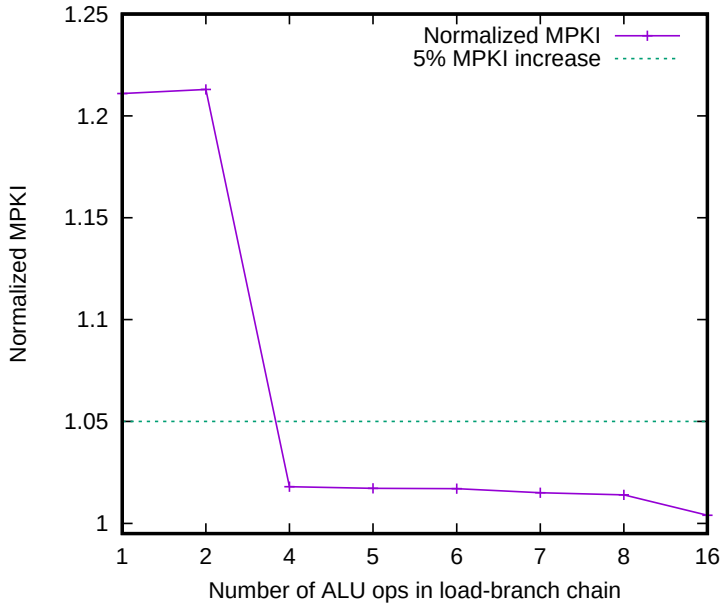

Figure 5.15: Most LDBP chains have 4 ALU operations between the branch and load(s).

entry track five operations. The total size of the CSB is 5-Kbit. The CST caches the backward slice of each branch. As there are 8 entries on the BOT, the CST must have 8 entries with 10 sub-entries (5 sub-entry for each branch source operand).

5.7 LDBP Sensitivity for FSM-ALUs

Figure 5.17 shows the sensitivity of 81-Kbit LDBP for different number of FSM-ALUs. The simulator infrastructure models a wakeup-select-like scheduling logic to choose FSM-ALUs when branch slices are ready to be pre-executed. The baseline for this study is the infinite LDBP. The MPKI drops by only 2% even if there is 1 FSM-ALU. A single FSM-ALU means that a LDBP branch branch with n ALU operations in the backward slice needs n cycles to complete. It also means that no other branch outcome can be processed in parallel. 1-2 FSM-ALUs are

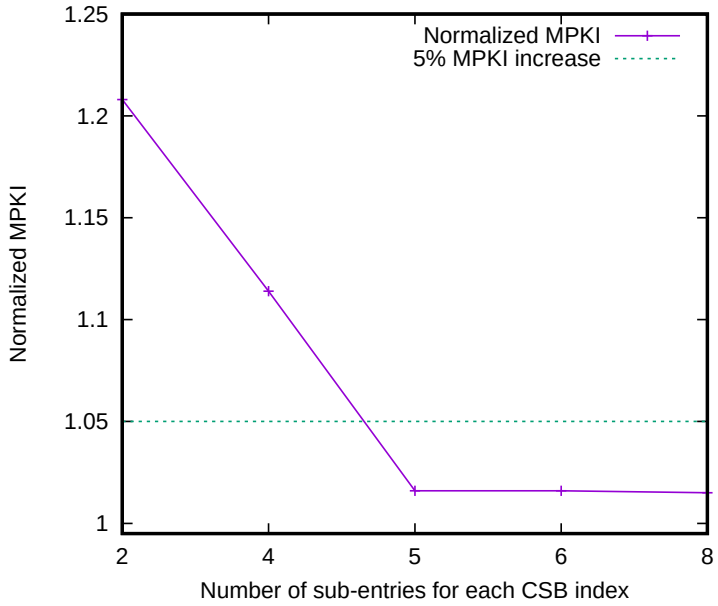

Figure 5.16: Each CSB index must have 5 sub-entries to capture LDBP backward slices.

sufficient to cater LDBP because of the timeliness of trigger loads (as discussed in Section 5.4. Also, a backward slice has just a few instructions between iterations. This means that it barely becomes the critical resource.

5.8 LDBP Gating and Energy Implications

The LDBP has significant performance gains, but some benchmarks (*libquantum*, *sssp*, *bc*) do not benefit. In this section, I will evaluate the effectiveness of gating the LDBP when infrequently used, to minimize energy dissipation.

Every component of LDBP is gated (low-power mode), apart from the Stride Predictor and RTT when there is a duration of 100,000 or more clock cycles where LDBP did not predict any branch. This phase is referred to as the LDBP low-power mode. As shown in Table 5.2,

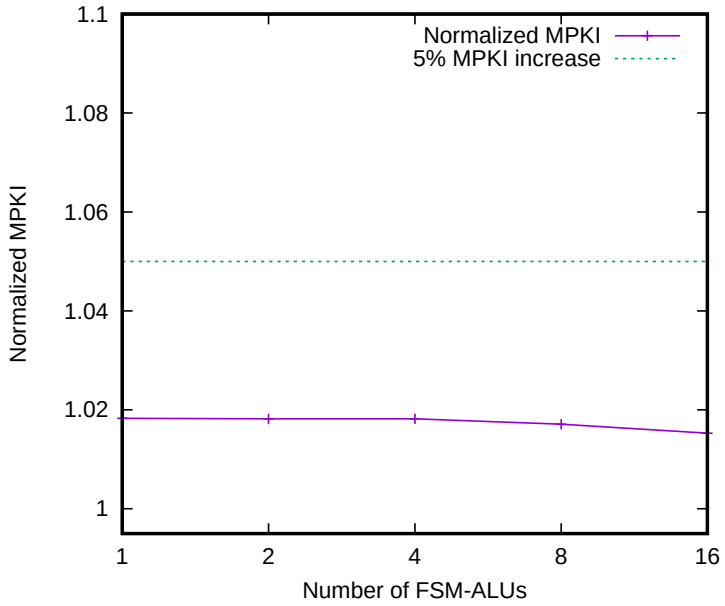

Figure 5.17: LDBP does not need many FSM-ALUs due to timely nature of trigger loads.

for *bc* and *sssp*, LDBP remains in low-power mode for 99.6% and 98.3% of the benchmark's execution time, respectively. Gating offers a considerable reduction in energy dissipated by LDBP as the predictor remains in low-power mode for 63.3% of the average execution time across all benchmarks, and LDBP gating does not have any negative effect on the prediction accuracy of LDBP.

Haj-Yihia et al. [14] present a detailed breakdown of core power consumption for high-performance modern CPUs running SPEC CINT2006 benchmarks. The thesis uses the data presented in their work to estimate the core energy dissipation. For this study's baseline energy model, the core power breakdown given in [14] is replicated for SPEC CINT2006 benchmarks. For the GAP benchmarks, the average power breakdown of SPEC CINT2006 benchmarks given in [14] is used. The broad-spectrum power model based on SPEC CINT2006 benchmarks is

Benchmark	% time in low-power mode
spec06_gcc	98.6
spec06_hmmer	0.0
spec06_astar	55.7
spec06_gobmk	33.8
spec06_omnetpp	95.2
spec06_mcf	73.4
spec06_sjeng	93.7
spec06_h264ref	96.8
spec06_bzip2	80.3
spec06_libquantum	99.9
spec06_perlbench	85.5
spec06_xalan	90.1
gap_bfs	3.4
gap_pr	2.7
gap_tc	0.1
gap_cc	32.4
gap_bc	99.5
gap_sssp	98.3

Table 5.2: Proportion of execution time in LDBP low-power mode.

good enough to capture the energy dissipation behavior of GAP benchmarks with a good level of accuracy.

Energy Per Access (EPA) for IMLI and LDBP were calculated using CACTI 6.0 [34]. For IMLI, an ideal structure with a single port is modelled. LDBP has 55% lesser EPA than IMLI even if it is assumed that all the tables are accessed when not in low power mode, which is not the case in reality. There is a 7% average increase in DL1 access for LDBP, which will result in an equivalent escalation in energy on the memory sub-system. The 12% decrease in MPKI when using LDBP will compensate for this increase in energy dissipation. Lesser MPKI implies lesser energy spent on executing the wrong branch path. The thesis does not account for the energy saved due to reduced wrong path execution in the LDBP energy estimation numbers.

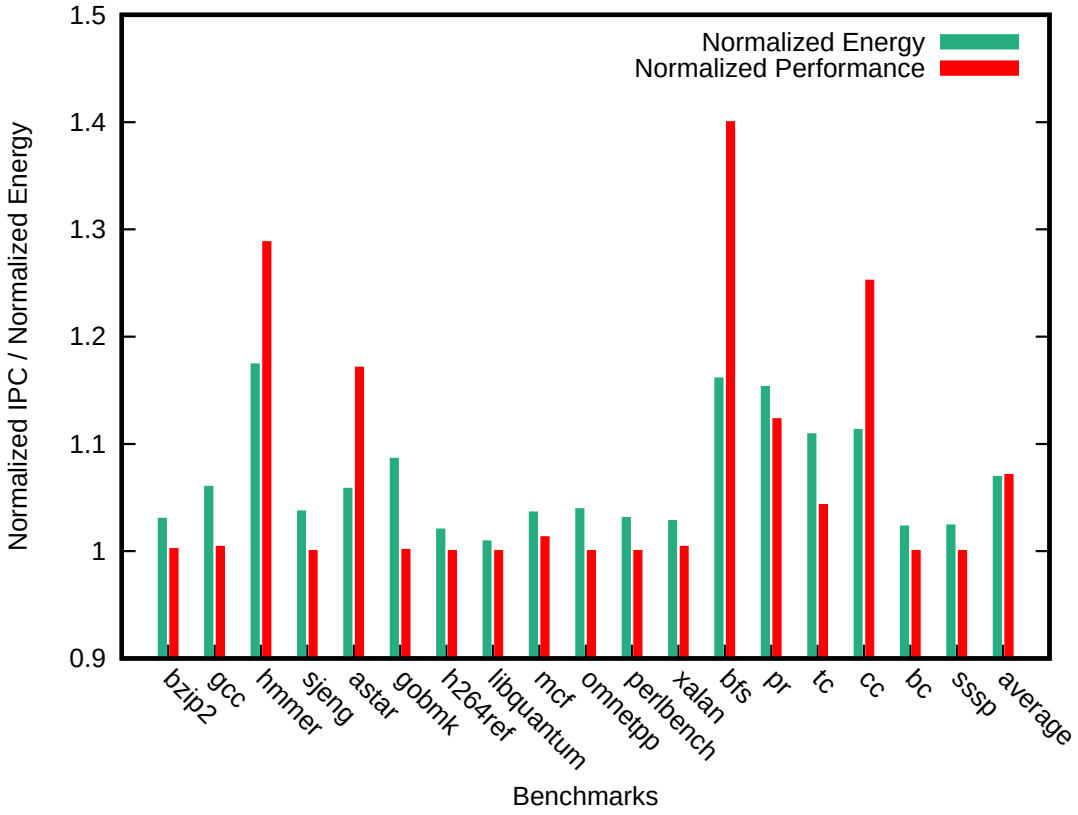

Figure 5.18: LDBP maintains a favorable energy-performance tradeoff.

Added to that, this study also does not account for the energy reduction incurred due to 7.14% lesser execution time when using LDBP. Reducing execution time results in reduced energy, and LDBP's pessimistic energy estimation model does not consider this.

Figure 5.18 shows the energy-performance tradeoff for IMLI + LDBP compared to the baseline 256-Kbit IMLI. The IPC boost outweighs the increase in energy dissipation for the majority of the benchmarks that benefit from LDBP. Benchmarks like *bc* and *sssp* only have about 2% energy overhead as the RTT and Stride Predictor continue to be active even under low-power mode. Interestingly, LDBP only predicts a negligible proportion of branches in *gobmk*, but it contributes to 8% more energy use. This is because LDBP resolves multiple low-frequency

branches that spread across different execution phases. Thus, *gobmk* does not offer a consistent low-power mode phase for LDBP. A more aggressive clock gating with retention state or smarter phase learning could further improve the *gobmk* case, but we leave it as future work.

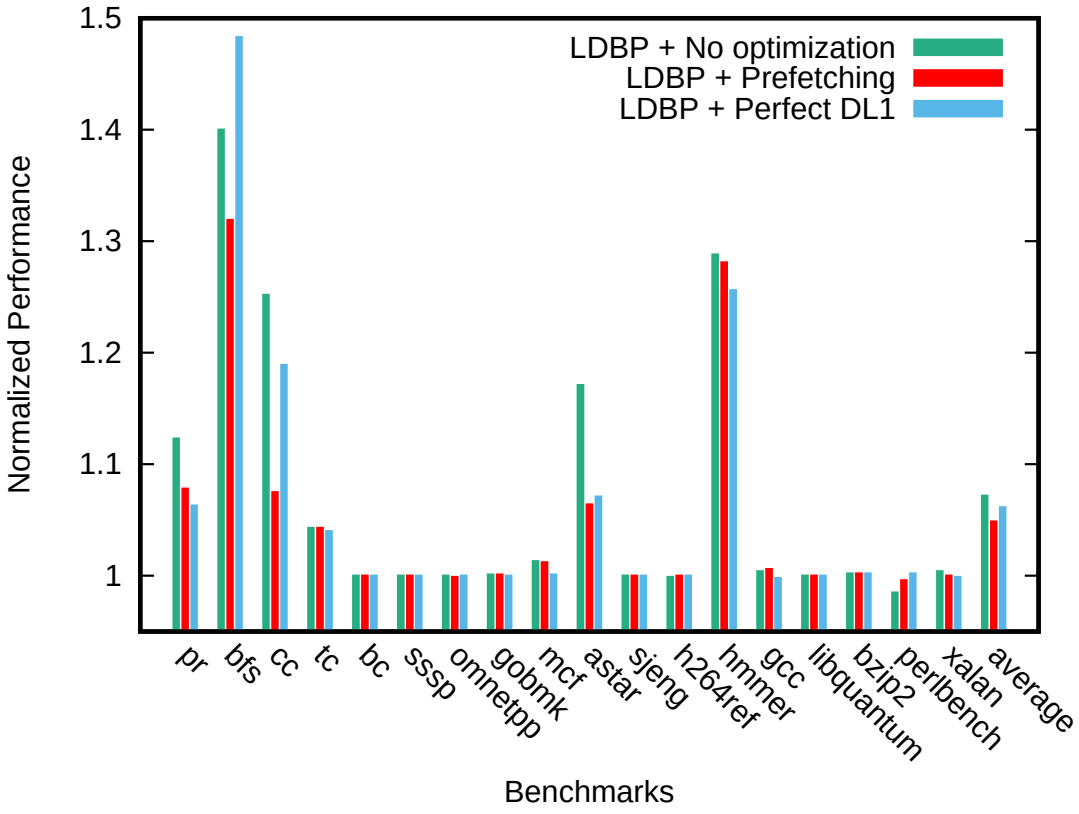

Figure 5.19: Triggering loads does not offer any unfair gains to LDBP.

5.9 Impact of Triggering Loads on LDBP Performance Gains

Figure 5.19 shows the normalized speedup of LDBP over 256-Kbit IMLI with three different Zen 2 core configurations. One, the default Zen-2 core used for evaluation in other parts of this dissertation. Two, the default core with a standard stride prefetcher and third, the default

core with perfect DL1 cache. It can be noticed that the IPC numbers are almost similar across all three configurations for most benchmarks. This clearly shows that prefetching trigger loads in LDBP do not provide an unfair advantage to it over the standalone IMLI predictor. Maybe even more important, Figure 5.19 shows that the LDBP benefits are consistent independent of memory sub-system improvements.

Chapter 6

Conclusion and Future Opportunities

> The important work of moving the world forward does not wait to be done by perfect men.
>
> George Eliot

6.1 Conclusions

As shown by the benchmarks evaluated as a part of this dissertation, branch outcomes dependent on arbitrary load data are hard-to-predict and contributes to a considerable proportion of mispredictions. They have poor prediction accuracy with current state-of-the-art branch predictors. These branch patterns are common in data structures like vector, maps, and graphs.

In this thesis, I propose the Load Driven Branch Predictor (LDBP) to eliminates the misses contributed by this class of branch. LDBP exploits the predictable nature of the address of the loads on which these hard-to-predict branches depend on and triggers these dependent

loads ahead of time. The triggered load data are used to precompute the branch outcome. With LDBP, programmers can traverse over large vectors/maps, do data-dependent branches, and still have near-perfect branch prediction.

LDBP contributes to minimal hardware and power overhead and does not require any changes to the ISA. The experimental results in this thesis show that compared to the standalone 256-Kbit IMLI predictor, the combination of 256-Kbit IMLI and LDBP predictor shrinks the branch MPKI by 12% and improves the IPC by 7.14%. The efficiency of LDBP also allows having a smaller primary predictor. A 150-Kbit IMLI + LDBP predictor yields performance improvement of 6.63% and 5.25% lesser mispredictions compared to the baseline 256-Kbit IMLI.

Another opportunity that this work provides is to extend the use of graphs further. As the GAP benchmark suite results show, LDBP can improve performance from graph traversals significantly. There is an extensive set of works exploring graphs for neural networks [53], for which LDBP could help to boost the performance.

6.2 Future Works

LDBP opens up the potential for exploring multiple opportunities on top of what is presented in this thesis. Below are some of the avenues which will be explored as a part of future work.

```
1 LOOP: addi a6, a6, 16
2       lw a5,4(a6) //donor load
3       .
4       .
5       .
6       lw a4,0(a5)//recipient load
7       bnez a4,LOOP
```

Figure 6.1: Sample code to explain a load-load chain. The source operand of *bnez* has a direct dependence with the recipient load and an indirect dependence with the donor load.

6.2.1 Load-Load Chains

As explained in Section 3.2, in load-load chains, the donor load may have a predictable address, but the recipient load does not have a predictable address. In the current LDBP or TAGE-like systems, the branch depending on the recipient load has a high misprediction. Figure 6.1 shows that the recipient load's address can be easily derived from the highly predictable donor's address.

The current LDBP architecture cannot handle this type of chain because neither the address nor the data of *lw* (line 6) are predictable. In the current LDBP setup, this kills the backward slice creation. Nevertheless, *lw* (line 2) has a predictable address. On checking the RTT entry for the recipient load's source register, LDBP knows that the load address comes from a predictable donor load data. To achieve this, the donor load must be added to the backward slice created. The main complication is that the donor loads must be triggered in a timely manner so that there is enough buffer to calculate the recipient's load address using the donor's data.

6.2.2 Multi-path chains

As explained in Section 3.6 and Section 5.3.3, load-branch chains having multiple paths or backward slices are present in the benchmarks evaluated. The current LDBP architecture cannot capture multi-path chains because the hardware and the timing overhead to track multiple slices per branch is enormous. Moreover, LDBP must also predict which slice will be used by successive instances of that branch. These branches are very rare, but still, as a part of future work, the focus will be on exploring other benchmark suites and figuring out the impact of such branches on the overall mispredictions.

6.2.3 Using a better load address predictor

The LDBP branch predictor architecture presented in this dissertation uses a simple stride-based load address predictor [10]. In LDBP, the stride predictor is used to predict trigger load addresses and prefetch trigger loads ahead of time. The stride predictor works based on a simple principle - the address of successive occurrences of a load differs by a constant. The address of future loads can be predicted using the below equation:

$$A_{n+1} = A_n + (A_n - A_{n-1}) \tag{6.1a}$$

$$A_{n+x} = A_n + (A_n - A_{n-1}) * x \tag{6.1b}$$

The hardware budget required for stride predictor implementation is trivial. In addition to that, the stride predictor does not add any overhead to the pipeline's critical path. However, it

cannot register load address histories where the delta is not consistent. For instance, a simple recurring load address delta pattern like $4, 4, 2, 4, 4, 2...$ cannot be captured.

As a part of future work, the plan is to replace the stride-based predictor with much more complex load address predictors like the correlated load-address predictor [5] or a TAGE-based address predictor like the EVES [45]. Doing so will resolve more hard-to-predict branches that are dependent on loads having short but repetitive address history. The tradeoff associated with switching to a more complex address predictor is the higher hardware budget and increased timing overhead.

Bibliography

[1] Jorge Albericio, Joshua San Miguel, Natalie Enright Jerger, and Andreas Moshovos. Wormhole: Wisely predicting multidimensional branches. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 509–520, USA, 2014. IEEE Computer Society.

[2] Ehsan K. Ardestani and Jose Renau. ESESC: A fast multicore simulator using time-based sampling. In *High Performance Computer Architecture, Proceedings of the IEEE 19th International Symposium on*, HPCA'13, pages 448–459, Washington, DC, USA, Feb. 2013. IEEE Computer Society.

[3] Ariful Azad, Mohsen Mahmoudi Aznaveh, Scott Beamer, Mark Blanco, Jinhao Chen, Luke D'Alessandro, Roshan Dathathri, Tim Davis, Kevin Deweese, Jesun Firoz, et al. Evaluation of graph analytics frameworks using the gap benchmark suite. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*, pages 216–227. IEEE, 2020.

[4] Scott Beamer, Krste Asanović, and David Patterson. The gap benchmark suite. *arXiv preprint arXiv:1508.03619*, 2015.

[5] Michael Bekerman, Stephan Jourdan, Ronny Ronen, Gilad Kirshenboim, Lihu Rappoport,

Adi Yoaz, and Uri Weiser. Correlated load-address predictors. *ACM SIGARCH Computer Architecture News*, 27(2):54–63, 1999.

[6] Geoffrey Blake, Ronald G Dreslinski, Trevor Mudge, and Krisztián Flautner. Evolution of thread-level parallelism in desktop applications. *ACM SIGARCH Computer Architecture News*, 38(3):302–313, 2010.

[7] Lei Chen, Steve Dropsho, and David H. Albonesi. Dynamic data dependence tracking and its application to branch prediction. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, HPCA '03, page 65, USA, 2003. IEEE Computer Society.

[8] Paul Chow and Mark Horowitz. Architectural tradeoffs in the design of mips-x. In *Proceedings of the 14th annual international symposium on Computer architecture*, pages 300–308, 1987.

[9] M Umar Farooq, Khubaib, and Lizy 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)*, pages 59–70. IEEE, 2013.

[10] John WC Fu, Janak H Patel, and Bob L Janssens. Stride directed prefetching in scalar processors. *ACM SIGMICRO Newsletter*, 23(1-2):102–110, 1992.

[11] Hongliang Gao, Yi Ma, Martin Dimitrov, and Huiyang Zhou. Address-branch correlation:

A novel locality for long-latency hard-to-predict branches. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pages 74–85. IEEE, 2008.

[12] Ed Grochowski, Ronny Ronen, John Shen, and Hong Wang. Best of both latency and throughput. In *Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings. IEEE International Conference on*, pages 236–243. IEEE, 2004.

[13] Saurabh Gupta, Niranjan Soundararajan, Ragavendra Natarajan, and Sreenivas Subramoney. Opportunistic early pipeline re-steering for data-dependent branches. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, pages 305–316, 2020.

[14] Jawad Haj-Yihia, Ahmad Yasin, Yosi Ben Asher, and Avi Mendelson. Fine-grain power breakdown of modern out-of-order cores and its implications on skylake-based systems. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(4):1–25, 2016.

[15] T. H. Heil, Z. Smith, and J. E. Smith. Improving branch predictors by correlating on data values. In *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 28–37, 1999.

[16] John L. Hennessy and David A. Patterson. *Computer Architecture; A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2007.

[17] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[18] Yasuo Ishii. Fused two-level branch prediction with ahead calculation. *Journal of Instruction-Level Parallelism*, 9:1–19, 2007.

[19] Daniel A Jiménez. Fast path-based neural branch prediction. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 243. IEEE Computer Society, 2003.

[20] Daniel A Jiménez and Calvin Lin. Dynamic branch prediction with perceptrons. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, pages 197–206. IEEE, 2001.

[21] Daniel A Jiménez and Calvin Lin. Neural methods for dynamic branch prediction. *ACM Transactions on Computer Systems (TOCS)*, 20(4):369–397, 2002.

[22] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203*, 2018.

[23] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[24] P. M. Kogge. *The Architecture of Pipelined Computers*. Hemisphere Publishing Co, New York, 1981.

[25] Chih-Chieh Lee, I-CK Chen, and Trevor N Mudge. The bi-mode branch predictor. In *Proceedings of 30th Annual International Symposium on Microarchitecture*, pages 4–13. IEEE, 1997.

[26] Chit-Kwan Lin and Stephen J. Tarsa. Branch prediction is not a solved problem: Measurements, opportunities, and future directions, 2019.

[27] Doug Matzke. Will physical scalability sabotage performance gains? *Computer*, 30(9):37–39, 1997.

[28] S. McFarling. Combining Branch Predictors. Technical Report TN-36, Jun 1993.

[29] Charles Melear. The design of the 88000 risc family. *IEEE Micro*, 9(2):26–38, 1989.

[30] Pierre Michaud. A ppm-like, tag-based branch predictor. *Journal of Instruction-Level Parallelism*, 7:1–10, 04 2005.

[31] Pierre Michaud, André Seznec, and Richard Uhlig. Trading conflict and capacity aliasing in conditional branch predictors. In *Proceedings of the 24th annual international symposium on Computer architecture*, pages 292–303, 1997.

[32] Dale Morris, Mircea Poplingher, Tse-Yu Yeh, Michael P Corwin, and Wenliang Chen. Method and apparatus for predicting loop exit branches, 2002.

[33] A. Moshovos, D. Pnevmatikatos, and A. Baniasadi. Slice-processors: an Implementation of Operation-Based Prediction. In *International Conference on Supercomputing*, pages 321–334, Sorrento, Italy, Jun. 2001.

[34] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. Cacti 6.0: A tool to model large caches. *HP laboratories*, 27:28, 2009.

[35] Harish Patil and Joel Emer. Combining static and dynamic branch prediction to reduce

destructive aliasing. In *Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6 (Cat. No. PR00550)*, pages 251–262. IEEE, 2000.

[36] Fred J Pollack. New microarchitecture challenges in the coming generations of cmos process technologies (keynote address). In *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, page 2. IEEE Computer Society, 1999.

[37] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. Efficient invisible speculative execution through selective delay and value prediction. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 723–735, 2019.

[38] Resit Sendag, J Yi Joshua, Peng-fei Chuang, and David J Lilja. Low power/area branch prediction using complementary branch predictors. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–12. IEEE, 2008.

[39] André Seznec. A case for (partially)-tagged geometric history length predictors. *Journal of InstructionLevel Parallelism*, 2006.

[40] André Seznec. A 64 kbytes isl-tage branch predictor. 2011.

[41] André Seznec. A new case for the tage branch predictor. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 117–127. ACM, 2011.

[42] André Seznec. Tage-sc-l branch predictors. 2014.

[43] André Seznec. Exploring branch predictability limits with the mtage+ sc predictor. 2016.

[44] André Seznec. Tage-sc-l branch predictors again. 2016.

[45] André Seznec. Exploring value prediction with the EVES predictor. In *CVP-1 2018 - 1st Championship Value Prediction*, pages 1–6, Los Angeles, United States, Jun. 2018.

[46] André Seznec, Stephen Felix, Venkata Krishnan, and Yiannakis Sazeides. Design tradeoffs for the alpha ev8 conditional branch predictor. *ACM SIGARCH Computer Architecture News*, 30(2):295–306, 2002.

[47] André Seznec, Joshua San Miguel, and Jorge Albericio. The inner most loop iteration counter: a new dimension in branch history. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 347–357. IEEE, 2015.

[48] John Paul Shen and Mikko H Lipasti. *Modern processor design: fundamentals of superscalar processors*. Waveland Press, 2013.

[49] Timothy Sherwood and Brad Calder. Loop termination prediction. In *Proceedings of the Third International Symposium on High Performance Computing*, ISHPC '00, pages 73–87, Berlin, Heidelberg, 2000. Springer-Verlag.

[50] James E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, ISCA '81, pages 135–148, Washington, DC, USA, 1981. IEEE Computer Society Press.

[51] David Suggs, Dan Bouvier, Michael Clark, Kevin Lepak, and Mahesh Subramony. Zen 2. `https://www.hotchips.org/hc31/HC31_1.1_AMD_ZEN2.pdf`.

[52] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanović. The risc-v instruction set manual, volume i: User-level isa, version 2.1. 2016.

[53] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A comprehensive survey on graph neural networks, 2019.

[54] T.-Y. Yeh and Y. N. Patt. Two-level adaptive training branch prediction. *24th Int'l Symp. on Microarchitecture*, pages 51–61, Nov. 1991.

[55] Tse-Yu Yeh and Yale N Patt. Alternative implementations of two-level adaptive branch prediction. *ACM SIGARCH Computer Architecture News*, 20(2):124–134, 1992.