Criticality Driven Execution

Aniket Deshmukh

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

TR-HPS-2025-001
June, 2025

The Dissertation Committee for Aniket Deshmukh
certifies that this is the approved version of the following dissertation:

Criticality Driven Execution

Committee:

Yale N. Patt, Supervisor

Mattan Erez

Poulami Das

Christopher J. Rossbach

Rustam R. Miftakhutdinov

Criticality Driven Execution

by

Aniket Deshmukh

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

May 2025

Acknowledgments

Many people contributed both directly and indirectly, in big and small ways, to help me reach where I am today.

Beginning, of course, with my family. My parents continue to be my biggest supporters in whatever path I choose. My mum, Anjali Deshmukh, is the closest confidant and friend I have ever had. She raised me to be loving and kind, open-minded and independent, to learn to live with myself through all my shortcomings and weaknesses. Our conversations - over everything from philosophy, science, technology, Yoga, and the like helped me grow and understand both myself and the world around me in a way I could never have managed on my own. I only hope that I can continue to support her in the best way I can and carry forward her spirit of giving back to society.

My baba, Aashish Deshmukh, has always been my strongest pillar of support. His dedication toward excellence and discipline was what drove me to perform to the best of my ability, no matter what the task. However, his love and care shone through in many moments - when he gave me my first Isaac Asimov book, when he helped me with fluid mechanics during my entrance exams, but most prominently: when I chose to do my PhD. I still remember him saying, "Take as long as you want, study all your life if you wish. We are here to support you emotionally and financially, so don't hold yourself back." These words gave me the confidence to push forward without fear, and I cannot thank him enough for that.

Toward my younger brother, Aneesh Deshmukh, I feel a great debt. We've had fun times together, but I feel I wasn't there for him when he was going through tough times during my undergraduate years, yet he was there to lend me an ear and listen to all my troubles over the course of my PhD. I can only hope you lean on me more for support. My cousin, Shivansh Dutt, ever a bundle of energy, made sure the time we spent over the years was always insane fun. I thank my aunt, Anuradha

Harke, and my grandmother, Shakuntala Harke, for always greeting me with a smile when I go back home to India and for all the amazing food I can never get anywhere else in the world.

I thank all my friends in Austin for their company over these past few years. Prateek, Wenqi, and Anyesha, who joined the PhD program alongside me, have been constant companions, both in the office at EER and outside. I thank Abbie, who was happy to chat over coffee anytime, and Jaegun, for being a good friend and driving me from time to time. I thank Divija, Jaeyong, Andrew, Margaret, and Rathna, among the juniors, for going along with anything I planned. Special thanks to Kayvan and Matthew for proofreading everything I gave them, and of course, for their great company, especially over the past summer.

I thank Ali and Kayvan, who helped me sound out my research ideas and listened to all my rants, Sophia, who was always willing to help out with everything and always had food for us, and Evan, who wasn't around for long but fit perfectly into our CBP team.

I haven't been in touch with folks back in India as much as I should have - Aditya, Masroor, and the rest of the Class-VIII group. But the few chances we did have to talk always reminded me of the good times.

During my internships, I met many people who helped me grow professionally and provided perspective on how the industry works. I thank Doug Carmean for giving me the freedom to explore as much as I wanted at Microsoft, and especially Rob Chappell, who helped me mold the core ideas for my dissertation early on. I thank Jayesh Gaur and John Combs for my time at Intel. I am grateful to Niket, Mo, Rustam, and Kulin for fully supporting me during my two final Apple internships and giving me the opportunity to work on engaging projects.

I enjoyed the advice and company of all the UT professors. I thank Mattan Erez for his honest opinion on everything, and Poulami Das for all her advice.

One of the biggest aspects of my time as a PhD student was being part of the HPS research group. Faruk often drove me home after our group meetings, and chatting with him about his hobbies and life in general was a lot of fun. Despite being the most senior student when I first joined, he made me feel truly welcome. There was never a dull moment with Ben around - his comments always kept things interesting, and his assistance with all my memory-related questions was invaluable.

In Stephen, I saw someone going through similar struggles to mine and coming out on top. His advice on how to conduct research and how to speak clearly to convey your ideas helped me through my middle PhD years, when I really struggled, and I am thankful to him for this. Siavash was always welcoming and happy to talk about whatever I wanted - research, personal life, or just to go get fried chicken, which I really appreciate.

The last seven years of my PhD wouldn't have been enjoyable without Chester. Ever since my first publication, the two of us have talked about research so often that we've probably memorized the intricacies of each other's work by now. Having someone with whom you can discuss everything: basketball, food, life in general, and almost everything else under the sky made me feel I wasn't in this alone, and I truly thank Chester for that. I also thank Leticia, who, as part of the staff, supported the group with all our administrative needs.

I thank all my PhD committee members—Mattan Erez, Chris Rossbach, Poulami Das, and Rustam Miftakhutdinov—for their support and feedback, which went a long way towards ensuring the dissertation was up to the mark. I also thank them for adjusting to what was a very hectic timeline in the last few months.

Finally, I thank Dr. Patt for all he has taught me over the last eight years. His ability to explain complex concepts so simply that even students fresh out of high school could understand stuck with me when I first TAed for him, and has always been a goal I wish to achieve someday. As a young PhD student who joined the group, he encouraged me to freely explore the breadth and depth of computer architecture

research, which helped me build a strong foundation. As his head TA, he showed me the importance of being sharp, precise, and meticulous in everything I did. These were also some of the best times I had teaching students, and I thank him for this opportunity. Above all, I thank him for his trust in me, which allowed me to achieve this milestone of completing my PhD dissertation.

-Aniket, April 2025, Austin, TX

Abstract

Criticality Driven Execution

Aniket Deshmukh, PhD
The University of Texas at Austin, 2025

SUPERVISOR: Yale N. Patt

Modern out-of-order (OoO) cores achieve high single-thread performance by maintaining a steady instruction supply through accurate branch prediction and reducing memory access latencies using data prefetchers and a cache hierarchy. Despite advancements in prediction algorithms and data prefetching techniques, the remaining branch mispredictions and cache misses still present major bottlenecks in many applications. Moreover, these bottlenecks often overlap in many applications - accelerating them together is vital for extracting the full benefit associated with solving these bottlenecks. Most prediction alternatives proposed in academia, like precomputation and runahead execution, only target either branch misprediction or cache misses and provide limited coverage. While solutions that target both branches and load like Slipstream exist, they require significant area, power, and energy investment.

This work provides a holistic approach for reducing the performance penalty associated with a significant fraction of these branch mispredictions and cache misses, without needing any additional execution hardware. Criticality Driven Execution (CDE) constructs accurate and lightweight dependence chains that issue early pipeline flushes for hard-to-predict branches and improve Memory Level Parallelism for long-latency loads simultaneously. It combines speculative precomputation—executing

specific chains twice—with instruction reordering that prioritizes the fetch and allocation of the remaining chains, accelerating the execution of these "critical" chains while maintaining high coverage. CDE dynamically redistributes existing OoO core resources to prioritize critical chains at the cost of delaying and providing fewer resources to "non-critical" instructions, achieving a 9% performance improvement without requiring a dedicated execution engine or separate OoO core.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

1.1 The Problem

Single-thread performance remains an important aspect of improving program runtime on out-of-order (OoO) cores. These cores require a steady instruction supply and fast memory accesses for high performance. The instruction supply is provided by a wide fetch unit coupled with an accurate branch predictor, while a multi-level cache hierarchy supported by data prefetchers reduces effective load latencies. However, despite decades of research, interruptions in instruction supply caused by branch mispredictions and backend stalls caused by cache misses that cannot be prefetched remain the two biggest limitations for OoO execution.

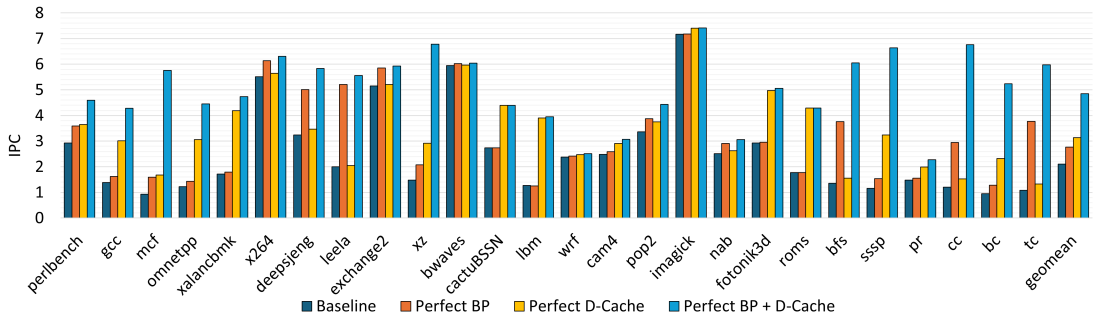

Figure 1.1: Eliminating all branch mispredictions and D-Cache misses

Figure 1.1 illustrates the potential performance gains if these bottlenecks are addressed in an 8-wide OoO core[1]. Eliminating all branch mispredictions (for both direct and indirect branches) improves Instructions Per Cycle (IPC) by 32%, and a perfect D-Cache provides a 49% IPC improvement. However, prior work has shown that these bottlenecks overlap in many applications [9]. Addressing them together reveals their true performance potential, and as seen in the final bar in Figure 1.1,

[1]Using a TAGE-SCL branch predictor and an aggressive Stream Prefetcher. The configuration for the baseline core is provided in Table 2.6

this provides a multiplicative speedup of 2.3x. Interestingly, a small subset of static branches and loads (∼256) is responsible for over 95% of branch mispredictions and load misses — these "hard-to-predict" branches and "long-latency loads" present key opportunities for improving performance.

Over the years, better branch prediction and load prefetching algorithms have been proposed to tackle this problem. However, the improvement in prediction mechanisms has plateaued recently, particularly for the set of hard-to-predict branches and load-latency loads (see Section 2.1). State-of-the-art academic proposals employing large neural networks that are too complex for hardware implementation only address ∼20% of the branch mispredictions [64] (over TAGE-SCL[51]) and only provide ∼30% D-Cache miss coverage [16] (over an aggressive stream prefetcher), leaving a lot of performance on the table. To bridge this gap, this dissertation focuses on mitigating the performance penalty associated with branch mispredictions and cache misses by directly addressing their impact on OoO cores.

1.1.1 Impact of Branch Mispredictions and Cache Misses

A branch misprediction forces the processor frontend to fetch wrong-path instructions until the branch is executed. Once executed, all instructions younger than the mispredicted branch are flushed, and the control flow is corrected. The longer the branch takes to execute, the more cycles wasted on fetching and processing wrong-path instructions. These wrong path fetch cycles are responsible for the bottlenecks associated with hard-to-predict branches, but can be decreased if these branches are fetched and executed faster.

Loads that miss in the Last-level Cache (LLC) take hundreds of cycles to complete, often causing full window stalls. This prevents subsequent instructions from entering the processor backend and beginning execution. LLC misses contribute to most of the performance penalty associated with cache misses[2] and their correspond-

[2]A perfect branch predictor and perfect LLC provide an improvement of 2.0x

ing stalls are primarily responsible for the bottlenecks associated with long-latency loads. These stall cycles can be reduced if more long-latency loads are initiated in parallel, which allows their execution latencies to overlap.

Thus, prioritizing the execution of hard-to-predict branches and long-latency loads over other instructions can help reduce their performance penalty. However, these branches and loads cannot be accelerated by themselves- instructions in their dependence chains are necessary for computing the corresponding branch predicates and load addresses. Mispredicted branches, loads that access memory, and instructions in their dependence chains are "critical instructions" as they primarily govern program performance.

1.1.2 Limitations of the Existing Execution Model

Modern processors fetch and allocate[3] instructions in program order, which does not consider criticality. However, there are more "non-critical" instructions than critical ones - over 85% of the dynamic instructions in the SPEC CPU2017 and GAP workloads on average. This larger proportion of non-critical instructions contributes minimally to program runtime but limits critical instruction throughput in two major ways:

Reduction in frontend bandwidth: Critical instructions are sparsely distributed and interleaved with non-critical ones. Consequently, younger critical instructions must wait for older non-critical instructions as fetch is performed in program order. This limits the effective frontend bandwidth for critical instructions, delaying how quickly they enter the processor backend and begin execution.

Fewer window resources: The instruction window from which an OoO core extracts parallelism is determined by the size of its Re-order Buffer, Physical Register File, Reservation Stations, and Load and Store Queues (window resources).

[3]Assigning Reservation Stations and Re-order Buffer entries to instructions after Rename

Critical instructions only occupy 26% of the out-of-order window on average during a full window stall (when more window resources are needed). This limits the amount of parallelism extracted from critical instructions, particularly Memory-Level Parallelism (MLP), which requires a large instruction window.

These limitations can be reduced by building a wider frontend to improve fetch bandwidth and a deeper backend to expose more parallelism. However, scaling the OoO core to enable this is expensive as a wider frontend increases pipeline latency [49] and a deeper backend increases area and power exponentially [43]. Moreover, scaling only increases the proportion of resources distributed to non-critical instructions.

1.2 Criticality Driven Execution

This dissertation introduces Criticality Driven Execution, a paradigm that prioritizes the fetch, allocation, and execution of critical instructions within the OoO core. Criticality Driven Execution (CDE) identifies critical instructions by tracing highly accurate (>99%) dependence chains for hard-to-predict branches and long latency loads at runtime. The chains provide over 80% branch misprediction and LLC miss coverage and are accelerated via two distinct execution models.

First, CDE combines hard-to-predict branch chains into an independent speculative precomputation thread that issues early pipeline flushes, addressing the frontend bandwidth limitation for these chains. Second, CDE prioritizes the fetch and allocation of long-latency load chains, utilizing backend resources more effectively to improve MLP for these chains. Finally, CDE combines these models to provide a unified solution that addresses both limitations together.

CDE does not use any additional window resources or Functional Units. Non-critical instructions are delayed and allocated fewer resources to prioritize CDE's chains. This allows for more efficient utilization of existing on-core structures without needing to scale the OoO backend or add additional execution hardware.

1.2.1 Identifying Critical Instructions

CDE identifies hard-to-predict branches and long-latency loads at runtime. It then traces their dependence chains, which are long and often span over 10,000 instructions. The chains are broken into basic block-sized segments and stored in a dedicated cache. These segments are stitched together at Fetch (using a decoupled branch predictor) to reconstruct the dependence chains on demand.

This mechanism creates dynamic chains that are accurate for any previously observed control flow, providing far greater branch misprediction and LLC miss coverage compared to prior work (which is limited to tracing chains within simple control flows). The reconstructed chains also contain synchronized timestamps that provide ordering relative to other non-critical instructions.

The chains for hard-to-predict branches and long-latency loads have different properties. Hard-to-predict branch chains generally have lower latencies and should be executed as early as possible to ensure the instruction stream is on the correct path. Long-latency load chains take hundreds of cycles to execute. While these chains benefit marginally from being initiated early, they primarily require a large OoO window so their loads can access memory in parallel. Given the different sources of benefit, CDE uses two different execution models.

1.2.2 Precomputation for Hard-To-Predict Branch Chains

CDE uses an independent, speculative precomputation thread consisting of hard-to-predict branch chains to reduce the misprediction penalty for these branches. This thread is fetched and executed faster as it contains fewer instructions than the full program and triggers early pipeline flushes using the timestamps generated while constructing the branch chains.

The "CDE precomputation thread" uses dedicated Fetch and Rename stages , but shares backend resources with the regular instruction stream (the "main thread"). These resources are freed as soon as possible to allow subsequent precomputation

thread instructions to enter the backend. Precomputation thread instructions are speculative and do not enter the Re-Order Buffer or commit their results. The CDE precomputation thread provides a 4.3% performance improvement over an aggressive baseline OoO core (8.3% for branch-intensive benchmarks). Its higher misprediction coverage allows it to outperform prior work; the CDE chains provide benefit if the precomputation result is ready **before the corresponding main thread branch is executed**, compared to prior work that can only use the precomputation result if it arrives **before the corresponding main thread branch is fetched**.

Since CDE's chains are highly accurate and timely, they provide high performance when executed on-core, even though this delays the main thread and executes two copies of all precomputation thread instructions (18.7% of the dynamic instructions on average). Using a dedicated execution engine only improves performance by 5.3%, showing that a sizable proportion of the backend resources used by non-critical instructions can be repurposed without hurting performance.

1.2.3 Preferential Allocation for Long-Latency Load Chains

To improve MLP, CDE dynamically reorders the instruction stream by preemptively fetching and allocating resources for long-latency load chains.

This "preferential allocation" model divides window resources - the Re-order Buffer (ROB), Reservation Stations, Load and Store Queues, and Physical Registers - into two partitions. The larger partition is assigned to long-latency load chains, replacing entries usually occupied by non-critical instructions. This allows multiple long-latency loads that normally cannot execute together (due to limited instruction window size) to reside in the backend simultaneously. Preferential allocation thus **expands the sequential window from which parallelism can be extracted for critical instructions** by skipping over the allocation of non-critical instructions.

Unlike the precomputation approach, the prioritized chain instructions commit their results to avoid the overhead of executing these chains twice. In-order retirement

is maintained by comparing the oldest instructions in each partition, facilitated by the timestamps generated during chain construction.

This approach improves performance by 6%[4], higher than prior work targeting MLP improvement, such as Runahead. Variants of Runahead execution can only discover MLP during full window stalls, which are limited in processors with larger instruction windows. Preferential allocation, on the other hand, proactively discovers MLP by grouping and executing highly accurate long-latency load chains and is energy efficient as no duplicate instructions enter the backend.

1.3 Building a Unified Execution Model

Improving the performance of both hard-to-predict branches and long-latency loads simultaneously requires prioritizing certain chains depending on their properties.

The simple approach - combining both types of chains into one large precomputation thread has several limitations. Combining these chains increases the density of the precomputation thread, causing significant backend contention. Supporting this bigger thread requires a separate core or dedicated execution engine with enough window resources and functional units (as many as the OoO core itself) to avoid backend contention. Prior work targeting branch and load chains simultaneously, like Slipstream [60, 59] and Speculative Multithreading [34], opted for this approach. The additional hardware roughly doubles the backend area and power. Moreover, a larger combined thread limit the effective fetch bandwidth for hard-to-predict branch chains in some applications. This reduces the benefit, even with a dedicated execution engine.

The preferential allocation model prioritizes critical instructions without executing them twice and does not increase backend contention. While preferential

[4]Preferential allocation with just load delays branch resolution, and thus needs to prioritize some branch chains as well. Without branch chains, the performance gain is 3.1%

allocation can provide some benefit with hard-to-predict branch chains, it is limited by ROB capacity and in-order retirement, and thus runs much slower than the precomputation thread.

Combining the two models provides a means to optimize for both timeliness (faster chains) and coverage. The precomputation thread mainly targets hard-to-predict branch chains and a few long-latency load chains that benefit from early initiation, keeping it lightweight and timely. Preferential allocation accelerates the remaining hard-to-predict branch chains and most long-latency load chains, maintaining high coverage while reducing backend resource contention. CDE uses a dynamic algorithm that decides how these chains are accelerated. This provides a 9.0% performance improvement without requiring a dedicated execution engine or separate core.

1.4 Contributions

The contributions of this dissertation are as follows:

- A runtime mechanism for tracing long and extremely accurate dependence chains for any branch or load.
- The CDE precomputation thread, that can issue early misprediction flushes for branches whose precomputation result arrives after the corresponding main thread branch is fetched (but before it is executed).
- A mechanism that preferentially fetches and allocates resources for long-latency load chains to improve MLP. Unlike precomputation, these chains commit their results, utilizing backend resources as effectively as possible.
- A unified model for accelerating hard-to-predict branch and long-latency load chains simultaneously. This involves assigning specific branch and load chains to the CDE precomputation thread, ensuring it remains lightweight and timely,

while preferential allocation accelerates the rest of the hard-to-predict branch and long-latency load chains, providing high coverage.

- An algorithm that partitions on-core resources for the CDE precomputation thread, the critical stream, and other non-critical instructions. This provides performance comparable to using a dedicated execution engine or separate core with lower energy and power overhead.

1.5 Thesis Statement

Criticality Driven Execution efficiently utilizes existing window resources to preferentially fetch, allocate, and execute critical instructions - issuing early misprediction flushes for hard-to-predict branches and improving Memory Level Parallelism for long-latency loads - thereby reducing the performance penalty associated with most branch mispredictions and cache misses.

1.6 Dissertation Organization

This dissertation contains seven chapters. Chapter 2 provides background on prediction and outlines relevant prior work. Chapter 3 covers the chain construction mechanism, explaining how hard-to-predict branches, long-latency loads, and their chain instructions are identified and stored. Chapter 4 talks about the CDE precomputation thread, explains its benefits, and provides a hardware implementation to support its execution. Chapter 5 introduces preferential allocation, detailing its key features and the micro-architecture support needed. Chapter 6 analyzes chain properties and provides a unified execution model for accelerating hard-to-branch and long-latency loads simultaneously. Chapter 7 concludes the dissertation.

Chapter 2: Background and Prior Work

2.1 Prediction Mechanisms

2.1.1 Branch Prediction

The branch mispredictions discussed in Section 1.1 are partly caused by branches with complex control flow patterns that are difficult to learn for commercially implemented prediction algorithms such as TAGE [51] and Perceptron [26], even with larger predictor tables. For instance, an infinite-sized TAGE-SCL [52] only provides ∼10% reduction [64] in branch mispredictions over a 64KB TAGE-SCL. State-of-the-art academic branch predictors such as BranchNet [64] and Whisper [28] attempt to predict these control flows by incorporating offline training techniques using sophisticated learning models. However, they only achieve an additional ∼10% misprediction coverage. The remaining branch mispredictions come from data-dependent branches whose targets and directions only correlate with input data, making them inherently challenging to predict with current algorithms.

2.1.2 Data Prefetching

Data prefetchers face similar problems associated with complex load address patterns and data-dependent load addresses. Even large neural data prefetchers [16, 55] only reach ∼30% D-cache miss coverage with ∼80% accuracy. Commercial products today use stream-prefetchers to deal with LLC misses [20, 25, 57] and use PC-based stride prefetchers [15] at the D-cache. State-of-the-art academic prefetching algorithms [41, 42, 44] only provide ∼5% more LLC miss coverage compared to a stream prefetcher (used in the baseline) and struggle to achieve higher coverage without increasing memory traffic significantly.

2.2 Precomputation

Precomputation uses dependence chains to compute branch directions and load addresses ahead of time. If this computation is faster than the main program, the precomputed branch directions override the conditional branch predictor and the precomputed load addresses issue prefetches to the memory subsystem.

2.2.1 Compiler Generated Threads

Early precomputation approaches relied on static analysis to identify frequently mispredicting branches, long latency loads, and their dependence chains, constructing helper threads that operated in a separate context while sharing on-core resources with the main thread. However, these threads often included many unnecessary instructions because runtime control flow tends to follow a limited set of paths, whereas their compile-time analysis accounted for all possible paths to ensure correctness. This led to bloated helper threads that executed too slowly to offer meaningful performance gains. Subsequent work used profiling to remove instructions corresponding to infrequently seen control and data flows, decreasing the helper thread size [9, 27, 31, 63, 65, 66]. However, profiling is not always representative and cannot capture phase behavior, which limits precomputation accuracy.

Lookahead execution [19, 30] similarly uses a "skeleton" or reduced version of the main program (created at compile time) to precompute branch directions and load addresses. It partitions on-core resources to make space for the lookahead thread, but has poor timeliness, similar to helper threads.

2.2.2 Runtime Precomputation Threads

Runtime-only approaches identify hard-to-predict branches, long-latency loads, and their dependence chain instructions dynamically. This reduces the number of instructions in the precomputation thread, improving its timeliness.

Iterative Backward Dataflow Analysis (IBDA) [8] identifies instructions in

long-latency load chains by tagging Register Alias Table (RAT) entries with the PC of the last instruction that writes to each register. However, IBDA cannot track memory dependencies, which are required for tracing chains across calls and returns. Moreover, it only captures a single level of the dependence chain every time the long-latency load is seen, limiting the overall length of the chain. Since IBDA filters the normal fetch stream using the identified PCs, it cannot fetch dependence chain instructions faster than the main thread.

Gupta et al. [21] use a similar technique for identifying hard-to-predict branch dependence chains that contain a single load, followed by a few arithmetic operations, ending at a branch. Tracing these short and simple chains produces timely results, but provides low misprediction coverage. DP-SSMT [10] uses a dataflow walk to trace the dependence chains for loads and branches. The generated thread uses trigger instructions to initiate computation and drive its control flow.

Branch Runahead [46], the prior state-of-the-art in branch precomputation, identifies lightweight and timely dependence chains in applications with simple control flows. However, it struggles with more complex control flow patterns (has lower coverage) and requires a dedicated execution engine to support the parallel computation of these chains. Chapter 4 contains an in-depth comparison against Branch Runahead.

2.2.3 Slipstream

Slipstream [61] and Dual-Core Execution [24] take the concept of precomputation to its limit by running a copy of the program on a separate OoO core. This "ahead thread" executes as fast as possible since it runs unhindered on the separate core without slowing the main thread.

In the most recent Slipstream proposal [59], the ahead thread is constructed at runtime by removing all control-dependent instructions for hard-to-predict branches, allowing it to leverage misprediction-level parallelism [35]. While Slipstream offers

improved coverage for both loads and branches, the ahead thread is heavy-weight (executing over 75% of the program's dynamic instructions) and requires a dedicated OoO core to maintain high throughput for the ahead thread. Using a separate core reduces performance as the delay associated with communicating precomputed branch directions and load addresses to the core running the main thread hurts timeliness. The additional core doubles the area and power overhead, and the extra instructions increase energy consumption. Chapter 6 evaluates the performance and energy of a Slipstream-like approach with the CDE dependence chains.

2.2.4 Using Precomputation to Resolve Branches Early

Prior work uses precomputation to override the branch predictor, providing no benefit if the corresponding main thread branch has already been fetched. This is because they use a unified queue or multiple per-branch queues to forward precomputed directions to the branch predictor. These queues are expensive as they buffer hundreds of predictions per branch and have multiple write ports, as several branches in the precomputation thread can finish executing together.

Implementing early resolution with these queues (in addition to overriding the branch predictor) requires support for simultaneous reads from multiple frontend pipeline stages. Alternatively, the in-flight branch queue and precomputed branch queues can be scanned in parallel to match precomputation thread branches to their main thread counterpart. This enables early resolution for branches that have entered the backend as well. However, both solutions increase the hardware cost and complexity of these queues as they require fully-associative lookups over large queue sizes (over 300 entries long). Farcy et. al. [17] use the scanning approach but do not discuss its implementation overhead. Gupta et.al. [21] only provide a few fixed flush points in the frontend. The cost of these queues also prevents prior work from precomputing branch targets for indirect branches.

CDE does not have this limitation as it uses synchronized timestamps: the

sequence number for a branch in CDE's precomputation thread is the same as the sequence number assigned to the corresponding main thread branch. This allows CDE to issue early misprediction flushes by reusing existing flush mechanisms. The details are covered in Chapter 4.

2.3 Runahead Execution

Loads that access memory generally take several hundred cycles to execute. Most load addresses cannot be precomputed early enough to hide the full memory latency. Runahead execution instead initiates memory accesses for future independent loads in parallel during full window stalls. This improves memory bandwidth utilization (if the load addresses are correct), providing most of the benefit associated with prefetching the load. However, traditional variants of Runahead are only active during full window stalls, limiting their effectiveness in workloads with fewer stall cycles. This effect has become more prominent over the years as the window size continues to grow.

The original Runahead work [37] executes all instructions in Runahead mode and cannot discover much MLP in processors with a large ROB. Later work [22, 38, 39] only executes load chains in Runahead mode and uncovers more MLP. However, these chains have low coverage, often produce inaccurate memory addresses, and are still limited by shorter full window stalls in many benchmarks.

"Decoupled" versions of Runahead [23, 40] execute load chains on a dedicated execution engine and are not limited to full window stalls. These are similar to precomputation-based techniques and have the same pitfalls: they use short chains with specific dataflow properties that work well for applications with simple control flows but struggle on more irregular applications. Chapter 5 contains a quantitative comparison against Runahead execution.

2.4 Compiler Solutions

Compilers can identify hard-to-predict branches and long-latency through profiling. They can rearrange code by unrolling loops and hoisting hard-to-predict branches or long-latency loads to compute them earlier in the program or to improve parallelism. This eliminates the hardware overhead of constructing and storing dependence chains at runtime. However, purely compiler-based solutions have a few major flaws. They rely on profiling, which is not always representative. Hoisting distance is limited by architectural register pressure [62] as the associated dependence chain instructions also need to be moved. Further, the optimal instruction ordering depends on detailed microarchitectural parameters that may not be available at compile time.

CRISP[33] is a lightweight compiler solution with minimal hardware support that profiles workloads in a data center environment and prioritizes hard-to-predict branch and long-latency load dependence chains in the scheduling logic. However, its benefit is limited as it only speeds up their execution by a few cycles.

Control-Flow Decoupling [53] uses the compiler to hoist the control-flow computation within a loop. The hoisted code inserts computed branch directions ahead of time into a hardware queue read by the rest of the instructions. This is challenging to do in the absence of loops or for loops with fewer iterations, as significant code duplication is required to account for all control flows leading to a branch.

2.5 Other Related Work

Speculative multi-threading [34] splits the program into speculative threads at compile time and executes these threads on a different core to improve Instruction-Level Parallelism (ILP). It leverages the compiler to find good points to parallelize code. These threads are not always useful since instruction criticality cannot be accurately computed at compile time. Balasubramonian et al. [5] use a similar "future

thread" that executes (but does not commit) on a partitioned section of the core and forwards register values to the main thread.

Long Term Parking [50] and Shelf OoO Execution [56] leverage instruction criticality to improve the efficiency of Reservation Stations, but cannot extract parallelism beyond the capacity of the ROB. Agarwal et al. [3] use instruction criticality and control independence to reduce in-order fetch bottlenecks. Continual Flow Pipelines [58] finds independent long-latency load chains in a short loop and uses a ROB-less architecture [4] to enable a larger instruction window for long-latency loads.

2.6 Baseline Out-Of-Order Core

Core	3.2GHz, 8-wide issue, 12 cycle FE latency, Support for MOV elimination 512 Entry ROB, 256 Entry Reservation Station, 16-wide retire 16 Execution Ports (6-ALU, 2-ST, 4-LD, 4-FP/VEC), Oldest-first scheduling (3-ALU with BR, 3-ALU with SHFT&MUL, 2-FP with MUL, 2-FP with DIV) 400 Physical Regs, 192 entry load queue, 128 entry store queue
Predictors	64KB TAGE-SC-L[51], 32-entry Fetch Queue History-based indirect branch predictor, RAS 1 taken per cycle, 8K entry BTB
Caches	32KB 8-way L1 I-cache (4-cycle access) 1R, 1W port (2 banks) 48KB 12-way L1 D-cache (4-cycle access), 64B lines (4R 2W ports) 512KB 16-way L2 cache (12-cycle access) 1MB 16-way LLC (18-cycle access), 32 MSHRs
Prefetcher	Stream Prefetcher, 64 Streams (always on), Feedback Directed Prefetching to throttle prefetcher
Memory	DDR4_2400R: 1 rank, 2 channels 4 bank groups and 4 banks per channel tRP-tCL-tRCD: 16-16-16

Table 2.1: Core parameters

The baseline OoO core evaluated in this dissertation uses the x86 ISA. It has an 8-wide frontend with a decoupled branch predictor [47] that can predict up to one taken branch or sequential instructions spanning 128B per cycle. Fetch addresses generated by the decoupled branch predictor are filled into a Fetch Queue. The Fetch unit uses Fetch Queue addresses to read up to two sequential cache lines from the Instruction Cache (I-Cache). The Decode, Rename, and Allocation stages handle up to

8 micro-operations (uops) per cycle, with 128-entry instruction buffers between Fetch and Decode. The frontend is 12 stages deep, and the minimum fetch-to-resolution latency for a branch is 15 cycles. The load-to-use latency for D-Cache (Data-Cache) hits is 5 cycles. The processor contains a unified Reservation Station with oldest-first scheduling. All the parameters are listed in Table 2.1.

Chapter 3: Critical Chain Construction

CDE identifies critical instructions at runtime in two steps. First, it detects hard-to-predict branches and long-latency loads via a counter-based mechanism. Then it uses these loads and branches as initiation points for a Backward Dataflow Walk to trace their dependence chains. The identified critical instructions are broken into basic block segments and stored in a Block Cache. The CDE precomputation thread and preferential allocation eventually use these entries to reconstruct the constituent hard-to-predict branch and long-latency load chains as needed. This mechanism was first proposed and later used in my published work[13, 14].

3.1 Marking Hard-To-Predict Branches and Long-Latency Loads

A small subset of static branches and loads (256) account for over 95% of branch mispredictions and LLC misses[1]. Prior work uses counter-based techniques [11, 12, 46] to track these hard-to-predict branches and long-latency loads. These counters capture branches and loads above a fixed accuracy threshold. I adapt these techniques to build two "Critical Count Tables" that mark all branches and loads above a specified MPKI (mispredictions/misses per kilo instructions) threshold instead.

3.1.1 Critical Count Tables

The Critical Count Tables are independent structures updated at retirement. The Critical-Branch Count Table tracks direction and target mispredictions for direct and indirect branches. It is an 8-way set-associative table containing 256 entries, indexed with the branch Program Counter (PC). Each entry in the table has a 3-bit saturating counter. An entry is created for a branch when it mispredicts, with its

[1]The numbers were averaged across Simpoints of length 200M each

counter value initialized to 1. Each subsequent misprediction at that PC increments the counter. A branch is considered hard-to-predict if it has an entry in the Critical-Branch Count Table and its counter value exceeds 1.

All counters in the table are decremented periodically. This ages out counters for branches below a specified MPKI threshold. For example, with a decrement period of 5000 (i.e., the counter value is decremented every 5000 instructions), counters for branches with MPKI below 0.2 will tend towards 0 as it is decremented more often than mispredicts are seen.

The Critical-Load Count Table works similarly. When a load misses in the LLC, an entry is created. The corresponding counter is incremented on subsequent LLC misses caused by the same load PC, and all counter values are periodically decremented.

Thresholding based on MPKI instead of accuracy (which prior work uses) more directly captures which branches and loads contribute to slowdowns within an instruction window. However, this effect is only prominent in workloads with larger footprints and does not have a noticeable impact on the SPEC CPU2017 and GAP benchmarks.

The table parameters determine the number of chains accelerated. This affects the coverage and timeliness of the execution models, and is evaluated in Section 4.6 and Section 5.5.

3.2 Identifying Dependence Chain Instructions

Dependence chains for marked hard-to-predict branches and long-latency loads are traced using a modified version of the Backward Dataflow Walk [10, 22]. For this, instructions are first collected in a post-retire buffer called the Fill Buffer.

3.2.1 Fill Buffer

The Fill Buffer holds up to 512 micro-ops (uops). Uops are added to the buffer after retirement and stored in program order. Retired branches and loads query the Critical Count Tables before being added to the Fill Buffer. An entry in the Fill Buffer contains the decoded uop, memory addresses accessed by the uop, whether it was marked a hard-to-predict branch or long-latency load (as set by the Critical Count tables), and a bit signifying that the uop is in the marked dependence chains (initially set to 0). This is summarized in Table 3.1. Note that the Fill Buffer operates at the level of uops to avoid decoding chain instructions when they are eventually used. **For simplicity, all the examples in this dissertation contain instructions with a single uop.**

PC	Decoded Uop	Load/Store Address	H2P BR	Long-Lat LD	Chain bit
32 bits	64 bits	32 bits	1 bit	1 bit	1 bit

Table 3.1: Fill Buffer entry

3.2.2 Backward Dataflow Walk

The Backward Dataflow Walk identifies the minimal set of instructions needed to compute the marked hard-to-predict branches and long-latency loads. It is initiated when the Fill Buffer is full. The rest of this section explains the chain identification process in the context of hard-to-predict (H2P) branches, but functions similarly for long-latency loads.

The example in Figure 3.1.a shows a code segment containing an H2P branch, with its assembly and control flow diagram. Figure 3.1.b shows how the Fill Buffer is populated when this code segment is executed and retired. The H2P branch is marked in red. When the Fill Buffer is full, the Backward Dataflow Walk begins. Starting at the youngest instruction, the Fill Buffer is traversed one entry at a time until an H2P branch is encountered. Registers and memory addresses needed to

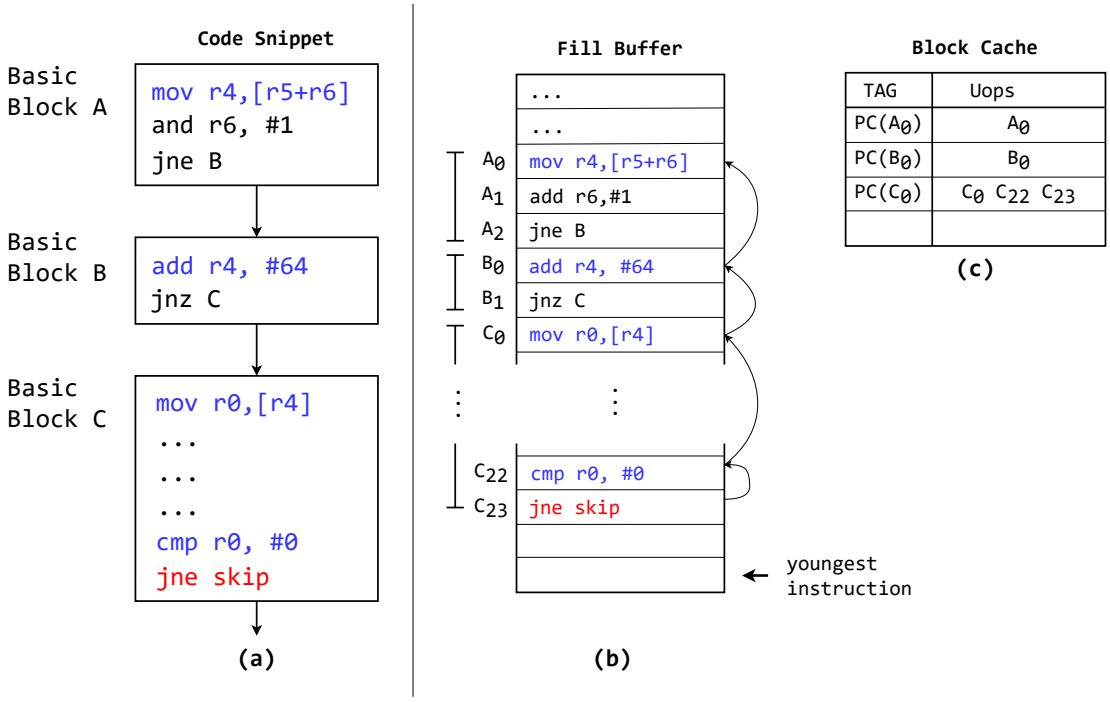

Figure 3.1: Identifying dependence chain instructions in the Fill Buffer

compute that branch are then added to a "Source List". In the example, C_{23} is the first H2P branch encountered during the Backward Dataflow Walk, and the condition code register (RFLAGS for x86) is added to the Source List.

Any instruction that writes to a register or memory location in the Source List is part of the H2P branch dependence chains. Thus, C_{22} is marked as a dependence chain instruction (blue). At the same time, the C_{22}'s destination register (RFLAGS) is removed from the Source List, and C_{22}'s source register (R0) is added instead. This ensures that the Source List tracks the minimum set of live-ins needed to compute the marked H2P branches.

Continuing upwards, C_0 is marked as a dependence chain instruction, and the Source List is modified to contain register R4 and memory location [R4]. The Backward Dataflow Walk continues until it reaches the oldest instruction in the Fill Buffer, marking all the instructions highlighted in blue. **Note that chains for all**

branches marked as H2P, including multiple dynamic instances of the same H2P branch, are traced simultaneously via this mechanism. In contrast, prior work [10, 22, 46] only used the Backward Dataflow Walk to trace dependence chains one branch at a time and terminated the Walk after seeing a second instance of the branch being traced, limiting their chains to short loops.

3.2.2.1 Hardware Implementation

Uops are initially added to the Fill Buffer after the Decode stage since the decoded uops are not available at Retire (these are overwritten on a misprediction flush). After retiring, uops populate the rest of the fields in their Fill Buffer entries. Memory addresses are read out when the corresponding Load or Store Queue entry is retired. Alternatively, after address generation for a uop is completed, the addresses with their corresponding sequence numbers can be saved in a separate buffer until retire (indexed by sequence number and does not require associative lookups).

The Backward Dataflow Walk takes ∼500 cycles and is managed by a state machine. Register dependencies are tracked using a bit-vector with one bit per-architectural register. Memory dependencies are tracked using a small 32-entry buffer that records load addresses. Together, these structures form the "Source List" mentioned earlier.

Instructions retired during the backward Dataflow Walk are discarded. The Fill Buffer thus only samples a portion of the retired instruction stream. Performance is not sensitive to the duration of the Backward Dataflow Walk, and the associated structures do not need multiple access ports to perform the walk faster or capture all retired instructions. The Fill Buffer size does not affect performance significantly (<0.5% change) as I use bit-masks to extend dependence chains, as explained in Section 3.5.

3.2.3 Storing Dependence Chain Instructions

The marked dependence chain instructions (including hard-to-predict branches, whose chain bit is also set) are split into basic block-sized segments and stored in a "Block Cache". Block Cache entries are tagged with the PC of the first instruction in that basic block (Figure 3.1.c). These entries are later stitched together using predictions generated by the branch predictor to reconstruct the dependence chains at fetch time. This is discussed in Section 3.7.

3.3 Tracking Memory Dependencies

The Source List tracks memory dependencies as correlated loads and stores affect chain accuracy. This is most commonly seen when an H2P branch is within a function body. In such cases, the function's input variables are often part of the dependence chain. The push and pop operations that communicate these variables need to be included in the dependence chain to trace instructions beyond the function boundary.

Unlike register dependencies, memory addresses corresponding to correlated load-store pairs can change over time. Thus, incorporating memory dependencies sometimes increases chain size without improving its accuracy. However, the overall improvement in the length and accuracy of the dependence chains makes up for this in most benchmarks.

3.4 Tracing Longer Dependence Chains

Longer dependence chains improve timeliness as they initiate the computation for the corresponding hard-to-predict branches and long latency loads earlier. However, the maximum chain length that can be traced is limited by the Fill Buffer size and the position of the H2P branch in the Fill Buffer. Using chain instructions as initiation points for the Backward Dataflow Walk (the next time they are seen in the

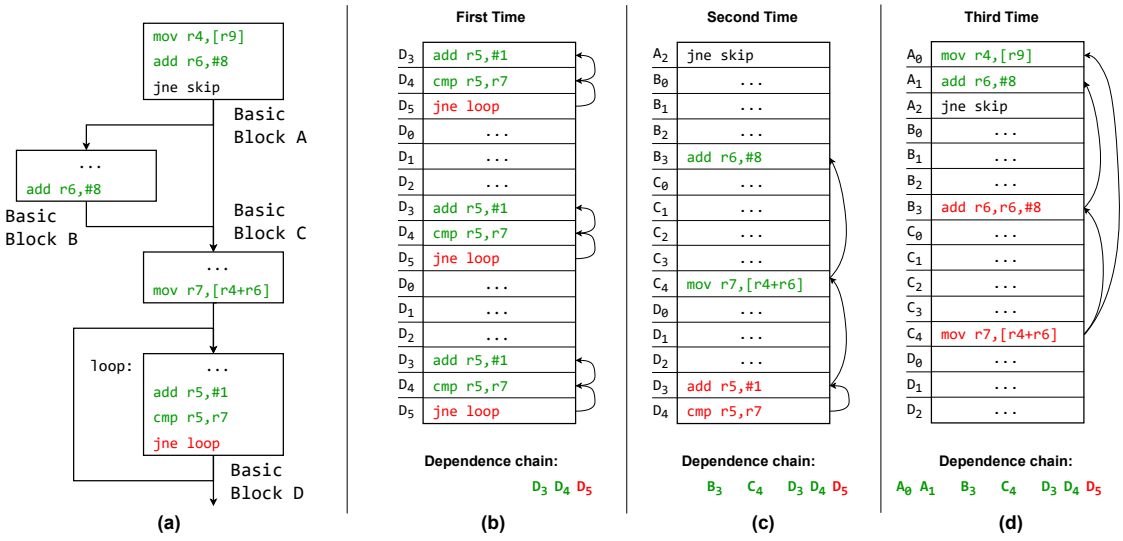

Figure 3.2: Tracing longer dependence chains using previously marked instructions
.

Fill Buffer) overcomes this limitation.

The example in Figure 3.2.a depicts this. This code snippet contains a loop branch that is H2P. Only a few instructions in the dependence chain of this H2P branch can be traced the first time it is seen in the Fill Buffer (Figure 3.2.b). The main thread keeps track of this information, and corresponding uops preemptively set their chain bit the next time they enter the Fill Buffer. Figure 3.2.c depicts a future Fill Buffer iteration. The Backward Dataflow Walk in this case is initiated at dependence chain instructions (highlighted in red this time) instead of an H2P branch. Additional instances of the Backward Dataflow Walk (Figure 3.2.d) trace more and more of the dependence chain. The individual basic block segments are then stitched together, which allows CDE chains to span thousands of instructions.

3.5 Combining Chains across Multiple Control Flows

An H2P branch can have different dependence chains for each control flow leading up to it. Figure 3.3 shows such an example. Under the control flow A-B-D,

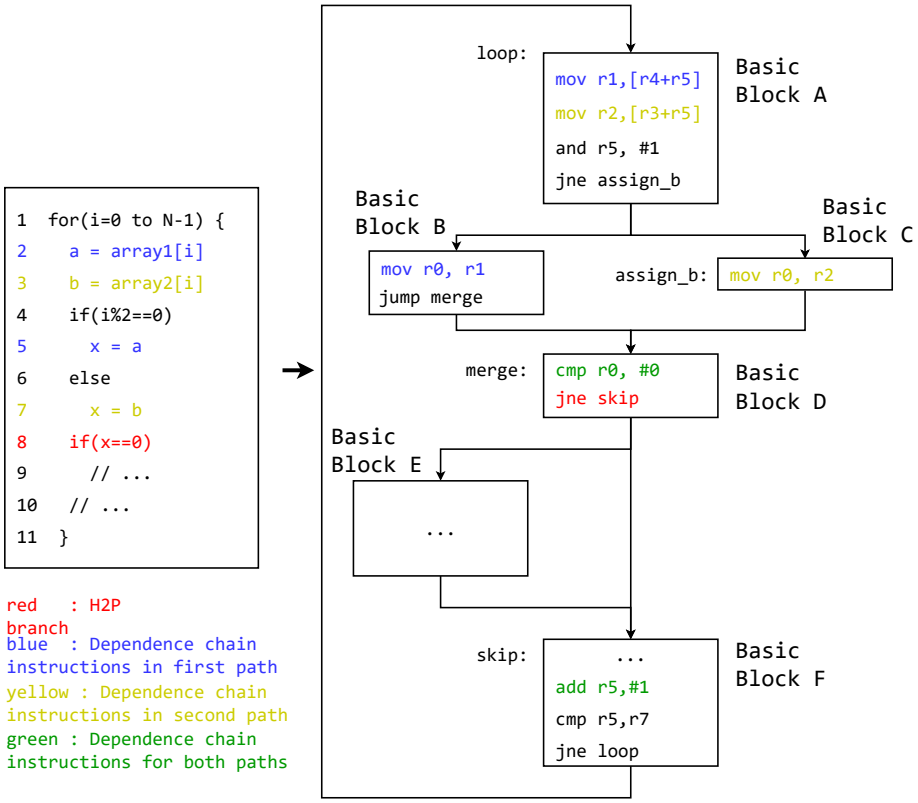

Figure 3.3: Tracing dependence chains across multiple control flows

the first instruction in basic block A is part of the dependence chain (in blue) for the H2P branch (in red). For A-C-D, the second instruction in basic block A is part of the dependence chain (in yellow). Saving the longest or the most recent dependence chain produces incorrect results if the saved chain does not match the actual control flow during fetch. Saving all possible chain versions is not viable as each additional branch added to the control flow (branch at the end of block A, for example) exponentially increases the number of possible chains.

Combining chains from multiple paths ensures the chain is correct across all these paths. This is done by storing a bit mask for each basic block that specifies which instructions in that basic block are part of H2P branch chains. Different versions of this bit-mask can be combined by bit-wise ORing them. In Figure 3.3, two masks are

generated for basic block A: 1000 for control flow A-B-D and 0100 for control flow A-C-D. The "1" in the bit mask indicates that the instruction is part of the chain. After bit-wise ORing, both the first and second instructions in basic block A are marked and saved in the Block Cache. This adds extra instructions (not needed to compute the H2P branch) to both paths but ensures the H2P branch chain is correct under both control flows. The masks are stored as part of the Block Cache entry for that basic block.

3.6 Steady State Operation

Figure 3.4 shows an overview of all the components. The Critical Count Tables are always active; retired branches and loads that access memory continuously update these tables independently of the Fill Buffer. Two additional bits in the ROB are added to track mispredicted branches and loads that miss in the LLC, which filter out the branches and loads that update the Critical Count Tables. In parallel, the Fill Buffer captures retired instructions (while querying the Critical Count Tables) until it fills up. When full, the Backward Dataflow Walk is initiated, and the identified chains uops are filled into the Block Cache.

Main thread instructions that were part of the CDE precomputation thread or the critical stream are marked with an additional bit in the ROB. This bit is carried along to the Fill Buffer and preemptively sets the chain bit in the Fill Buffer. This allows the Backward Dataflow walk to be initiated at previously marked chain uops, tracing longer chains as discussed in Section 3.4.

3.6.1 Block Cache

Uops with their chain-bit set after the Backward Dataflow Walk are divided into basic block segments and stored in the Block Cache. The Block Cache has 512 entries and is divided into a tag store and a data store. The tag store holds a 40-bit tag (PC of the first instruction in the basic block). The data store contains decoded

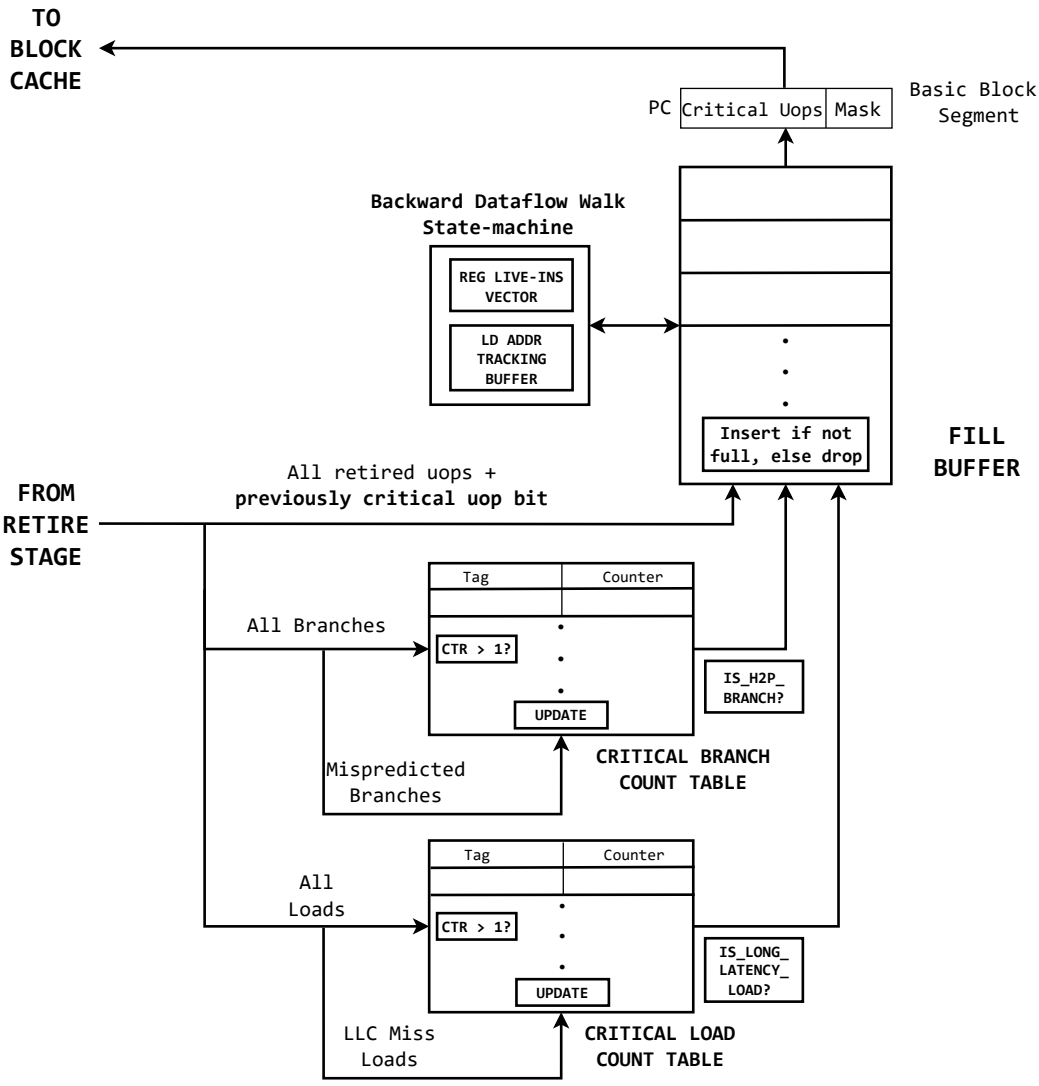

Figure 3.4: Implementation overview: Tracing dependence chains

dependence chain uops (4B per uop on average), the bit-mask (32 bits long), and the number of non-chain uops skipped (which are used for computing timestamps at Fetch).

On a tag match, the existing, older basic block segment is read out. Its bit-mask is ORed with the bit-mask for the incoming, newer basic block segment. The

corresponding uops are combined, and this combined basic block segment and bit-mask are then written to the Block Cache. Note that the fill operation is off the critical path and does not impact cycle time. This combines chains across multiple control flows as discussed in Section 3.5.

In most benchmarks, over half the basic block segments contain no chain uops for hard-to-predict branch and long-latency load chains. To optimize storage capacity, a smaller 256-entry tag store is reserved exclusively for basic block segments with no dependence chain uops. These do not need data store entries as they contain no chain uops and their bit-masks are zero. This tag store captures cases where a dependence chain traverses intermediate basic blocks with no chain uops and directs the Fetch unit not to terminate the dependence chain, as there may be more chain uops past the empty basic block segments.

Periodically resetting the bit-masks: Some control flow patterns are only observed during specific execution phases. Dependence chains captured with older control flows may not be valid after a phase change and therefore need to be removed to keep the chains lightweight. This is done by periodically resetting the bit-masks in the Block Cache and overwriting the existing basic block segments. The periodic reset ensures that future Backward Dataflow Walks do not use the bit-mask instructions as initiation points if they are no longer critical in the current execution phase. A reset period of 500K instructions worked best for performance.

3.7 Reconstructing the Dependence Chains at Fetch

Fetch addresses generated by the decoupled branch predictor in the baseline OoO core are forwarded to the Block Cache, which has a dedicated Fetch unit coupled to it. The Block Cache retrieves the corresponding basic block segments at these addresses and stitches them together to reconstruct the stored dependence chains.

Figure 3.5 shows how this works for the example in Figure 3.1. The control flow diagram for the code and corresponding Block Cache entries are repeated in

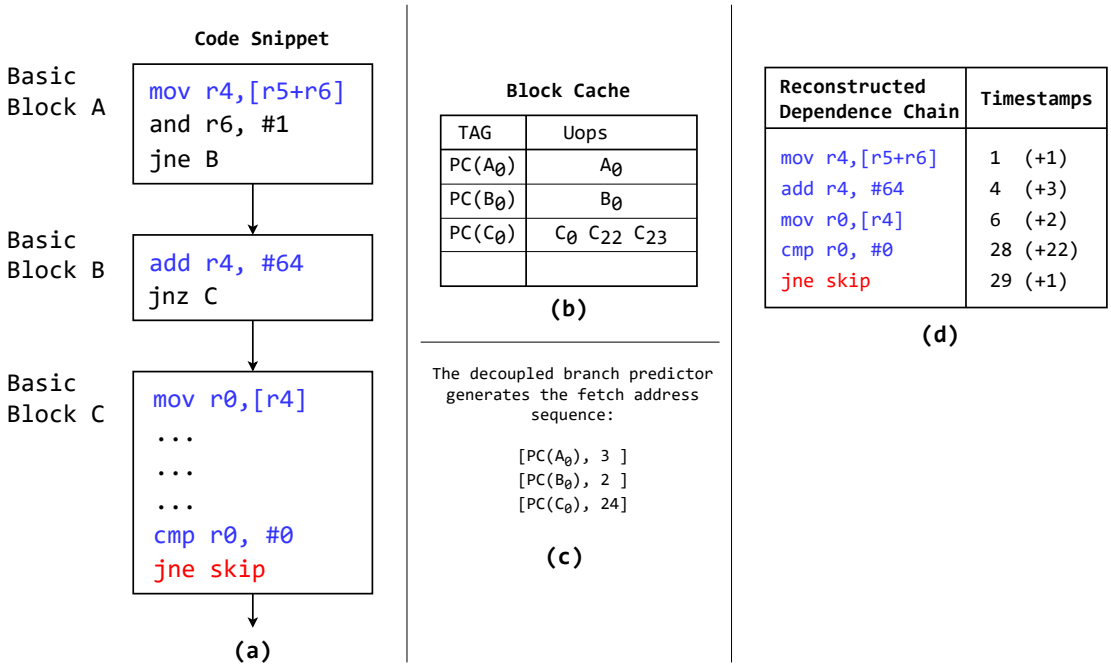

Figure 3.5: Fetching uops from the Block Cache

Figure 3.5.a and Figure 3.5.b. After the Backward Dataflow Walk completes, the program control flow reaches the same code snippet again, as it is part of a loop or a function that is called repeatedly[2].

The baseline decoupled branch predictor uses the current program counter (PC) to generate the next address at which instructions are fetched and the number of instructions to be fetched at that location. For this code snippet, the fetch addresses generated are shown in Figure 3.5.c. This takes 3 cycles, as the branch predictor has a throughput of one taken branch per cycle.

These fetch addresses are sent to the I-Cache (for the main thread) and the Block Cache in parallel. On a hit in the Block Cache, the CDE precomputation thread (or the critical stream in case of preferential allocation) is initiated. The chain uops

[2]Note that CDE can only improve a hard-to-predict branch or long-latency load if its dependence chain is seen multiple times

corresponding to the fetch address are read out of the Block Cache in 3 successive cycles and stitched together to reconstruct the stored dependence chain as seen in Figure 3.5.d. Regular fetch for this example would require more than 3 cycles as the instructions in basic block C cannot be fetched in a single cycle.

The number of skipped non-chain uops stored in the Block Cache is used to compute the timestamp for the chain uops. The computation is done in parallel with the rest of the Fetch stage processing and does not impact the critical path. This approach of capturing and stitching together basic block segments is similar to prior work on trace caches [18, 45, 48], except we only do this for critical instructions.

Chapter 4: Speculative Precomputation for Hard-To-Predict Branch Chains

Prior work has shown hard-to-predict branch chains usually have shorter latencies compared to other instructions and run faster when executed independently. To speed up their computation, I combine hard-to-predict branch chains into an independent, speculatively executed thread that precomputes branch directions.

4.1 CDE Precomputation Thread

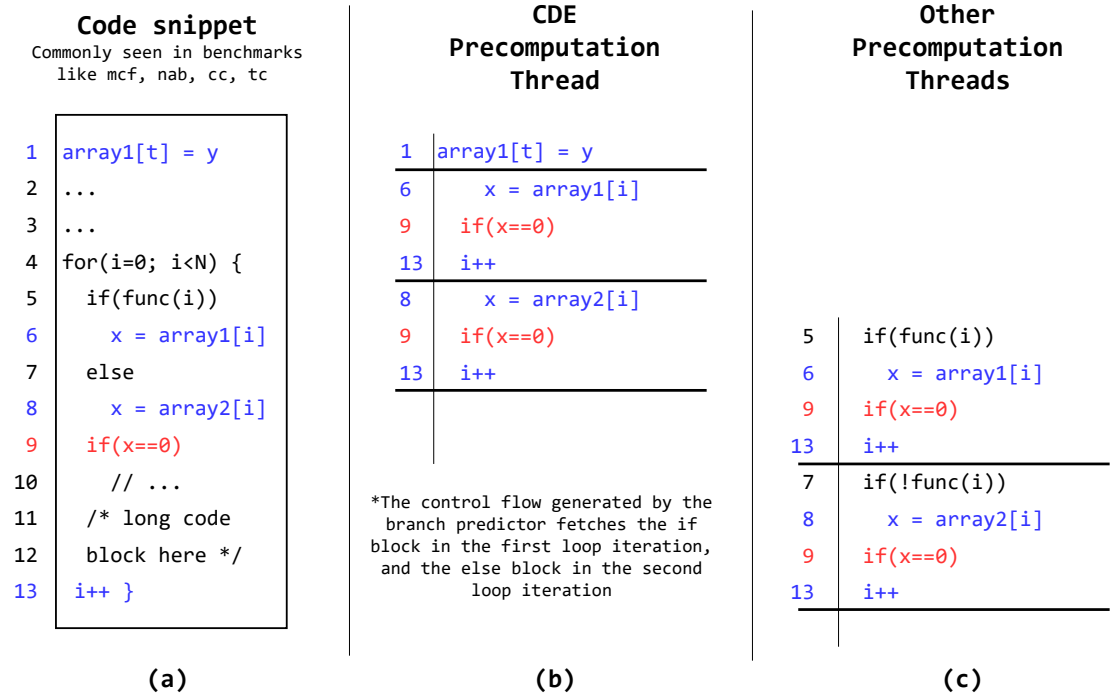

Figure 4.1: CDE precomputation thread example

The example in Figure 4.1 shows how the CDE precomputation thread works. The code snippet in Figure 4.1.a contains a hard-to-predict branch on line 9 (highlighted red). The control flow generated by the branch predictor fetches and executes

the if code block in the first loop iteration, and the else code block in the second loop iteration.

The CDE precomputation thread, shown in Figure 4.1.b, is constructed by stitching together basic block segments at fetch addresses generated by the branch predictor as explained in Section 3.7. It contains fewer instructions compared to the main thread and can thus be fetched faster (limited only by the branch predictor throughput). Given enough resources, the CDE precomputation thread executes multiple instances of the hard-to-predict branch and issues the corresponding misprediction flushes (if any) a few cycles before the main thread. The earlier the precomputation thread starts, the more opportunities it gets to run ahead of the main thread, saving more cycles of misprediction penalty.

When a precomputation thread branch is found to be mispredicted, the existing infrastructure uses the branch timestamp (generated during chain reconstruction) to issue a pipeline flush. This flushes instructions younger than the mispredicted branch in both threads. Fetch for both threads then resumes at the same point, i.e., their states are synchronized. The precomputation thread starts running ahead again, looking for the next misprediction. Thus, **the CDE precomputation thread reduces the time between successive mispredictions.**

4.1.1 Benefits of using the Main Branch Predictor

Non-H2P intermediate branch direction can affect which instructions are part of hard-to-predict branch chains. The branch on line 5 in Figure 4.1 is one such example. Prior work attempts to deal with these branches by either assuming the majority direction (which decreases chain accuracy) or by also precomputing their dependence chains (which adds many additional instructions, decreasing timeliness). Most recent work, including the prior state-of-the-art, Branch Runahead, chose the later approach (Figure 4.1.c).

The CDE precomputation thread, on the other hand, does not need to compute

intermediate branch directions. It uses fetch addresses generated by the main branch predictor (TAGE-SCL) to stitch together basic block segments that comprise the traced dependence chains. TAGE-SCL predicts non-H2P intermediate branches with very high accuracy (>99.9%) and the chains reconstructed using these predictions are thus also accurate. TAGE-SCL also predicts many hard-to-predict branches with over 80% accuracy, allowing the precomputation thread to correctly fetch instructions past them without waiting for the H2P branch to resolve.

Some prior work fetches branch chains faster than the CDE precomputation thread as they either ignore or statically predict intermediate branch directions, whereas the CDE precomputation thread has to wait for the main branch predictor to generate fetch addresses. This only works if the intermediate branches are biased. Prior work can also deal with intermediate branches if the chains are control independent, i.e., the dependence chain is unaffected by the direction of these intermediate branches [35, 46]. This works well in benchmarks where simple control flows are common, but provides very little performance in applications with more complex control flows as the opportunities for exploiting control independence are limited. The CDE precomputation thread performs well on both classes of applications as it can capture some of the benefit associated with control independence when TAGE predicts these intermediate branches accurately.

Traversing complex control flows also leads to longer chains. As seen in Figure 4.1, the CDE precomputation thread traces dependence chain instructions beyond the loop boundary and starts precomputation much earlier. It thus provides benefit even if the first instance of the hard-to-predict branch mispredicts. Other precomputation mechanisms either start after an initial misprediction is observed or at deterministic initiation points like the loop boundary. This flexibility allows the CDE precomputation thread to provide coverage far beyond what prior work achieved ($\sim$75% for CDE vs $\sim$20% for prior work).

4.1.2 Load Prefetching Effect

Some benchmarks have many long-latency loads within hard-to-predict branch chains. The CDE precomputation thread speeds up both types of chains in these benchmarks, providing some of the multiplicative benefit described in Section 1.1. This is evaluated in Section 4.6.3. However, the benefit is limited due to backend contention - an important observation that helps construct a unified model to address hard-to-predict branches and long-latency load simultaneously, which is discussed in detail in Chapter 6.

4.2 Implementation Overview

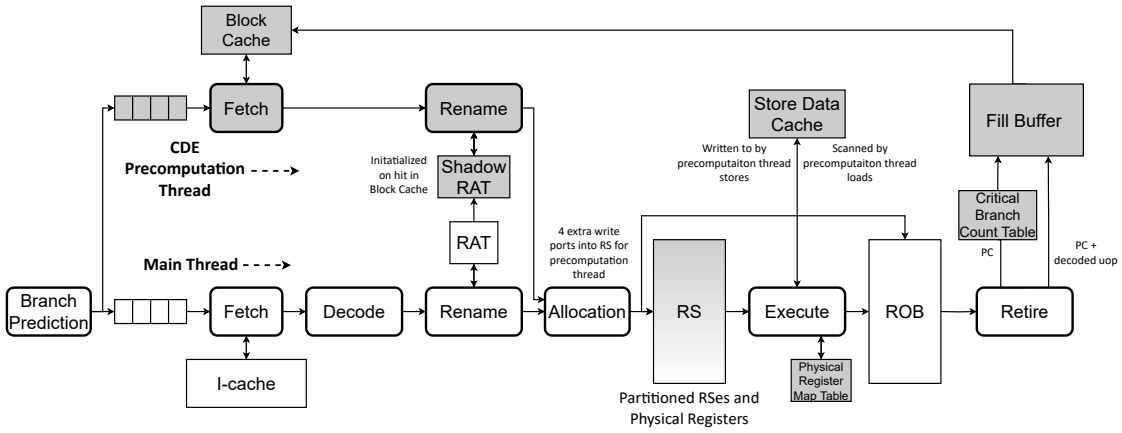

Figure 4.2: Implementation overview: Precomputation for hard-to-predict branches

An overview of the additional hardware required to construct, fetch, and execute the CDE precomputation thread is shown in Figure 4.2. The Critical-Branch Count Table and Fill Buffer located in the backend are responsible for identifying hard-to-predict branches and tracing their dependence chains. The precomputation thread has dedicated 8-wide shadow Fetch and Rename Stages. The shadow Fetch stage uses the fetch addresses inserted into a shadow Fetch Queue by the decoupled branch predictor to stitch together basic block segments from the Block Cache.

The Fill Buffer and Block Cache operate on uops instead of instructions. This

reduces the overall frontend latency for the CDE precomputation thread to 9 cycles. A shadow RAT manages precomputation thread dependencies. The precomputation thread and the main thread converge at Allocation, with 4 dedicated allocation ports added for the precomputation thread, while the remaining 8 are shared between the two threads. A fixed partition of Physical Registers and Reservation Stations is reserved for the precomputation thread when it is active (when inactive, these entries are used by the main thread). Precomputation thread instructions do not enter the ROB, and their Physical Registers are freed using a separate Physical Register map table. The precomputation thread has a small Store Data Cache for buffering store values. The parameters for all these structures are summarized in Table 4.1. The next few sections provide detailed descriptions for all the components other than the Fill Buffer and Critical-Branch Count Table (previously covered under Chapter 3).

Core	Dedicated 8-wide Fetch and Rename Stages, 9-cycle latency Allocation-ports shared with the main thread, 4 extra ports for precomputation thread 256 PRs, 128 RSes reserved for precomputation thread when active
Caches	Critical-Branch Count Table: 256-entry cache, 0.2KB, 1-cycle access, Dec period: 10k Block Cache: 512-entry cache, 256 zero-tags, 8-way, 19KB Store data cache: 64 entry cache (32B per entry)
Other structures	Fill Buffer: 512-entry queue, 8KB, single access port PR map table, 2400 bits, Shadow RAT

Table 4.1: Structure sizes for the CDE precomputation model

4.3 Frontend

The shadow Fetch stage is connected to the Block Cache and is accessed every cycle. On a hit, the precomputation thread is initiated. The precomputation thread fetches instructions at a higher rate than the main thread and thus consumes fetch addresses faster. To account for this mismatch, the Fetch Queue for the main thread is larger than a normal Fetch Queue (can hold up to 128 fetch addresses). This buffers more fetch addresses for the main thread, allowing the Branch Predictor to run further ahead and match the precomputation thread's throughput.

4.3.1 Fetch

To support 8-wide fetch from the Block Cache, basic block segments belonging to a single cache line are stored in different ways with the same cache-line index, similar to the Block-Based Trace Cache [7]. Segments longer than 8 sequential uops are divided into multiple entries. The shadow Fetch unit reads out all Block Cache entries in two consecutive cache lines for a given fetch address. Combining these sequential segments allows the Block Cache to deliver up to 8 uops per cycle. More efficient trace cache designs, such as those proposed by Patel et.al. [45], can be adopted to improve the Block Cache's throughput.

The bit-masks for the read-out Block Cache entries are added to a small queue that feeds the main thread. This marks uops in the main thread that are part of the precomputation thread so they can be used as initiation points for the Backward Dataflow Walk in the Fill Buffer after retirement.

4.3.2 Rename and Allocation

When the precomputation thread is initiated, the contents of the main RAT are copied into the shadow RAT to synchronize the state of both threads. After renaming, instructions are sent to the allocation logic, which prioritizes precomputation thread instructions. The dedicated frontend, preferential access to allocation slots, and partitioned backend ensure that the precomputation thread is not blocked due to back pressure caused by main thread instructions.

4.4 Backend

The precomputation thread is extremely accurate (less than 0.7% of its branches are incorrectly computed) and timely ($\sim$ 75% of precomputation thread branches save at least one cycle of misprediction penalty). Thus, prioritizing the allocation of precomputation thread instructions performs well, even though it delays the execution of main thread instructions.

A maximum of 128 Reservation Stations and 256 Physical Registers are given to the precomputation thread when active, allocated on a first-come-first-served basis. When inactive, all Reservation Stations and Physical Registers are given to the main thread. The partition is managed by two registers - one for Reservation Stations and one for Physical Registers, which keep track of the number of precomputation thread instructions using these resources. Instructions from both threads share Execution units, cache ports, and MSHRs (Miss Status Handler Registers). Precomputation thread instructions are identified by an additional bit in the Reservation Stations and are discarded when they finish execution. The scheduling logic is oldest-first (based on when they enter the Reservation Stations), which naturally prioritizes precomputation thread instructions as they are given allocation priority.

4.4.1 Freeing Physical Registers

The precomputation thread only maintains a speculative RAT. Its instructions free up backend resources as soon as possible to avoid the in-order retirement bottleneck. Precomputation thread instructions use a separate table containing a Valid bit and a 5-bit Reference Counter per Physical Register (PR) for identifying PRs that can be freed. These are initialized to 1 and 0, respectively, when the precomputation thread is initiated. When a precomputation thread instruction is renamed, it sets the Valid bit for the previous PR mapped to its destination Architectural Register (AR) to 0. If this previous PR is not currently being used (Valid=0, Reference Counter=0), it is freed.

Every instruction that wants to read from a PR increments its Reference Counter at Rename. The counter is decremented when the instruction reads the data value for that PR, just before it enters the Execution Units. After decrementing the counter, if it is also invalid (Valid=0, Reference Count=0) because a younger instruction overwrote its mapping, the PR is freed. This works because no instructions in the Reservation Stations need to read this PR, and instructions in the frontend

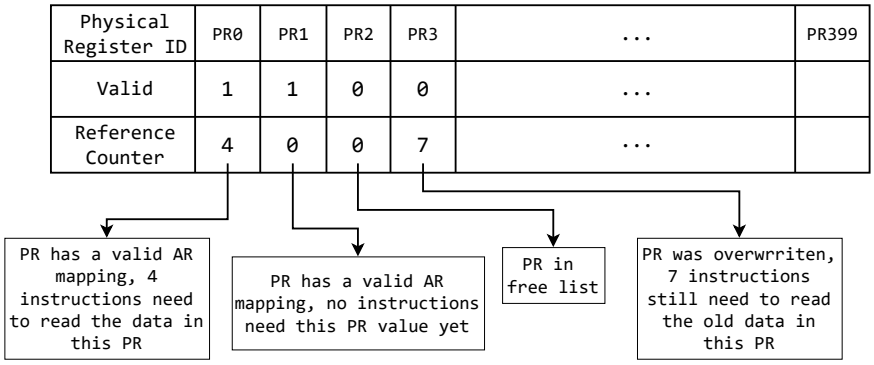

Figure 4.3: Physical Register Map Table

will use the new mapping for that AR. Figure 4.3 shows what various combinations of the Valid and Reference Counter bits specify.

This approach is similar to prior reference counter-based mechanisms for Renaming [4, 36], but does not incur the overhead of stalling execution to prevent counter overflows. If the counter overflows, the precomputation thread frees a Physical Register before all its consumers begin execution (this does not affect the main thread), and may lead to incorrect execution. However, this is rare since the precomputation thread is frequently flushed and does not impact precomputation thread accuracy much.

4.4.2 Dealing with Stores

Precomputation thread instructions do not change the processor's architectural state. Loads in the precomputation thread are treated as prefetches and are not allocated Load Queue entries. They leave the Reservations Stations when they have a D-cache port and carry their destination PR in the corresponding MSHR entry. However, precomputation thread stores cannot write to the D-cache as they update the machine's architectural state. Instead, they write to a small side cache that buffers the last 64 half-lines (32B) written to by precomputation thread stores. Precomputation thread loads read data from both this cache and the D-cache.

4.4.3 Branch Misprediction Flushes

When a mispredicted branch that is part of the precomputation thread finishes execution, a flush operation is triggered. Precomputation thread branches have the same timestamps as their main thread counterparts (computed during Fetch), and this timestamp is used to flush all instructions younger than the mispredicted branch (both the main thread and precomputation thread instructions). The processor backend is partially flushed exactly as it would be in a normal OoO core, and the checkpointed or recovered state of the RAT is copied over to both the main RAT and the shadow RAT. This, along with fixing the Branch Predictor history and PC, synchronizes the state of both threads.

Partially Flushing the Frontend: Normally, when a branch misprediction is detected in the backend, all the frontend pipeline stages are flushed. However, CDE supports partial flushes in the processor frontend, i.e., instructions older than the mispredicted branch in the frontend pipeline stages are not flushed. This situation arises when the precomputation thread runs so far ahead that the main thread branch being flushed is in the frontend. In this case, some main thread instructions in the frontend are older than the mispredicting branch and should not be flushed. This is enabled by adding a comparator before the flush signal for each pipeline stage, which compares the timestamp of the oldest instruction in that stage and the mispredicting branch. The comparator latency overlaps with other tasks performed during the flush operation, such as fixing the branch predictor history, and does not impact the overall misprediction flush latency. The Fetch Queue is also partially flushed - when this happens, the full misprediction penalty for that branch is saved. Note that when the frontend is partially flushed, the state of the main RAT does not need to be recovered. Only the shadow RAT needs to be fixed, and this is achieved by checkpointing the contents of the shadow RAT instead of the main RAT when the precomputation thread is running far ahead of the main thread.

When a precomputation thread branch resolves, the in-flight branch queue

entry (which tracks information for all in-flight main thread branches) for the corresponding main thread branch is modified to reflect the precomputed branch direction and target. When the main thread branch finishes execution, it reads the in-flight branch queue to check whether the misprediction was resolved correctly. If the precomputation was incorrect, another misprediction flush is issued to correct the control flow. However, this is rare (< 0.001 PKI) and its impact on performance is negligible.

4.4.4 Terminating the CDE Precomputation Thread

The precomputation thread is terminated on a miss in the Block Cache or when a precomputation is incorrect. On a miss, the remaining instructions continue to precompute branch directions until all remaining precomputation thread instructions have been executed. When an incorrect precomputation is detected, all younger precomputation thread instructions are drained from the processor.

Incorrect precomputations can be detected in two ways. The first examines the precomputed branch directions and targets saved in the modified in-flight branch queue entry (as explained above) and serves as a fail-safe. The second uses the bit-masks carried along with the main thread instructions.

When the precomputation thread is initiated, a poison bit in the main RAT is initialized to 0 for all ARs. Main thread instructions not part of the H2P branch dependence chains (have a 0 in the bit-mask) set the poison bit for the AR they write to. Main thread instructions that are part of the dependence chains (have a 1 in the bit mask) clear the poison bit for their destination AR. If a main thread instruction that is part of the H2P branch dependence chains (has a 1 in the bit-mask) reads from a poisoned register, it is flagged as incorrect. This is because reading from a poisoned register implies that the dependence chain instruction needed a result produced by a non-dependence chain instruction (after the precomputation thread started), which is incorrect behavior. Following this, precomputation thread branches that have not been executed yet and are younger than the instruction that caused this dependence

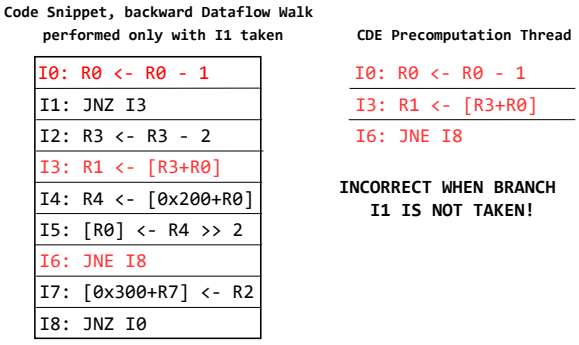

Figure 4.4: Example of an incorrect precomputation thread

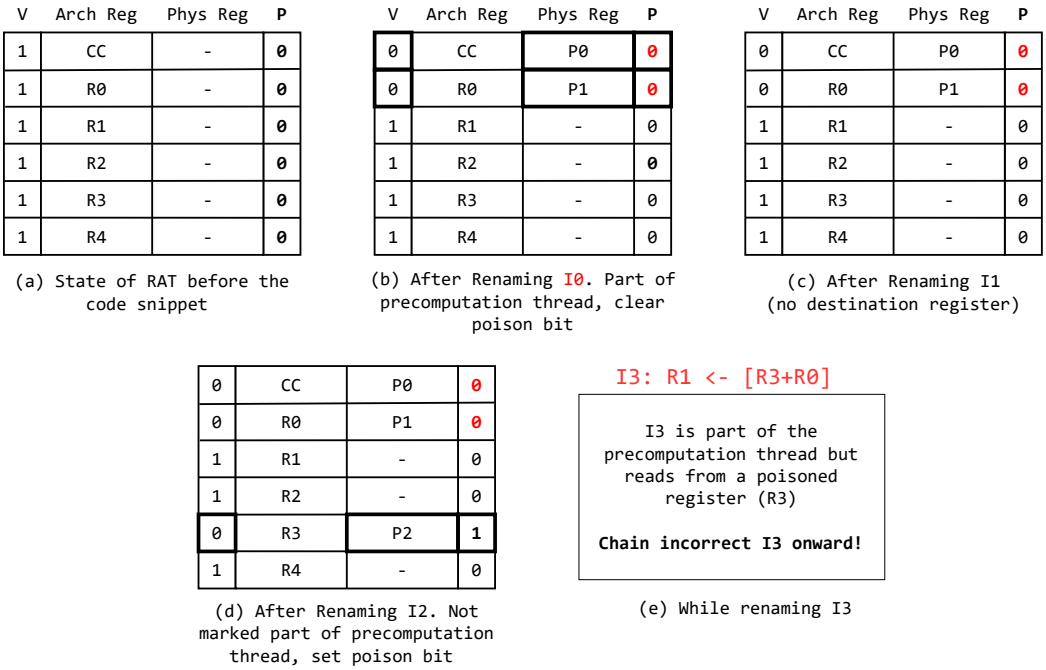

Figure 4.5: Sequence of events leading up to the detection of an incorrect chain

violation are blocked from triggering misprediction flushes. Precomputation is terminated, and the rest of the precomputation thread instructions are gradually drained from the backend. This helps filter out most incorrect precomputations without requiring a second flush, reducing the number of additional misprediction flushes to below 0.001 PKI (per 1K instructions).

Figure 4.4 contains an example showing an incorrectly constructed precompu-

tation thread. The Backward Dataflow Walk for this code snippet was performed only with branch I_1 being taken. This excludes instruction I_2 from the dependence chain. However, if during subsequent execution, branch I_1 is not taken, the dependence chain is incorrect. Precomputation results produced by this chain are likely incorrect until the new control flow is recorded in the Fill Buffer. The register poisoning scheme preemptively detects such scenarios and terminates the precomputation thread. The sequence of events involved in the detection is shown in Figure 4.5. These terminations occur with a frequency of less than 0.01 PKI, and thus have no noticeable impact on performance.

Late Termination: The precomputation thread is also terminated if its computation is "late". This happens when the precomputation result arrives in the same cycle that the main thread branch is executed, or after the main thread branch is executed. In either case, no misprediction penalty is saved. If only late precomputations are seen within the last 100K retired instructions, precomputation is turned off for 1M (retired) instructions. This primarily occurs when the precomputation thread contains most instructions in the program (but is allocated fewer Reservation Stations and Physical Registers).

4.5 Hardware Overhead

The CDE precomputation thread increases dynamic instruction footprint by 18.7% on average. This is much lower than Slipstream [59] (85%), which targets both load and branches, and is comparable to Branch Runahead [46] (34%). Since it executes instructions on-core rather than using a separate execution engine or core, the CDE precomputation thread is more efficient than prior work. McPAT[32] is used to estimate area, power, and energy consumption.

Area: The main area overhead comes from the Fill Buffer and the Block Cache. The duplicated pipeline stages and shadow RAT also add to this. This is approximately 3.5% of the total core area, with over 2% coming from just the caches

and Fill Buffer. Comparatively, a true 16-wide OoO core is much more challenging to implement: it costs ∼10% more area while only providing 2.1% performance.

Power: The additional structures increase peak power by 8.5% according to McPAT. The additional frontend is responsible for over 6% of the additional power, particularly the Block Cache, Rename stage, and the shadow RAT.

Energy: The increased dynamic instruction footprint leads to increased Fetch, Rename, and backend energy in addition to the energy consumed by the new structures. However, this is mitigated by the reduction in overall execution time and the reduction in wrong-path instruction fetches for the main thread. According to McPAT, the overall energy consumption in our evaluated benchmarks reduces by 1.8%.

4.6 Evaluation

4.6.1 Methodology

I use Scarab[1], an execution-driven cycle-accurate x86-64 simulator, to simulate the micro-architecture of an aggressive OoO core augmented with the additional structures and logic needed for CDE. Main memory is modeled using Ramulator[29].

Benchmarks I use SPEC CPU2017 benchmarks[2] with the ref input set and the GAP benchmarks suite[6] (with inputs g=19 and n=300) in all my evaluations. The SimPoint[54] methodology is used to find representative regions and generates up to 5 Simpoints per benchmark, with 200 million instructions per Simpoint. Each run is preceded by a warmup period of 200 million instructions.

4.6.2 Performance

Figure 4.6.b shows the per-benchmark performance improvement for the CDE precomputation thread. The geomean improvement is 4.3%. The branch MPKI (both direct and indirect branch mispredictions) for these applications is shown in Figure 4.6.a, sorted in decreasing order of branch MPKI. Figure 4.6.c shows the

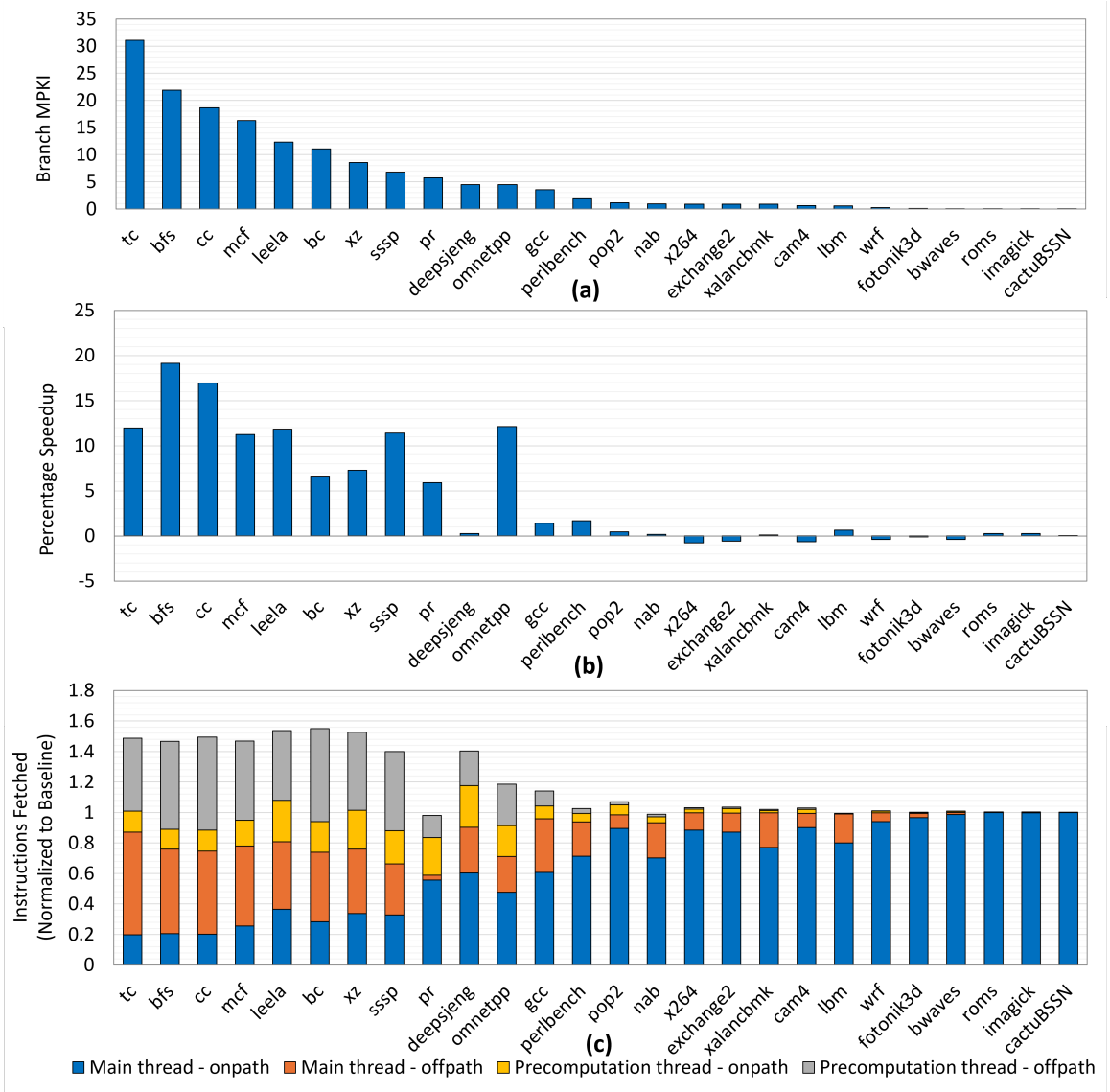

Figure 4.6: CDE precomputation thread: Branch MPKI, speedup and percentage of instructions

number of uops fetched from each thread relative to the baseline.

The best-performing benchmarks: *tc*, *bfs*, *cc*, *mcf*, and *leela* have many mispredictions, most of which are on the critical path of the program. *Omnetpp* and *sssp* have a relatively lower branch MPKI but show substantial improvements as they have many long-latency loads in H2P branch chains. This accelerates both types of

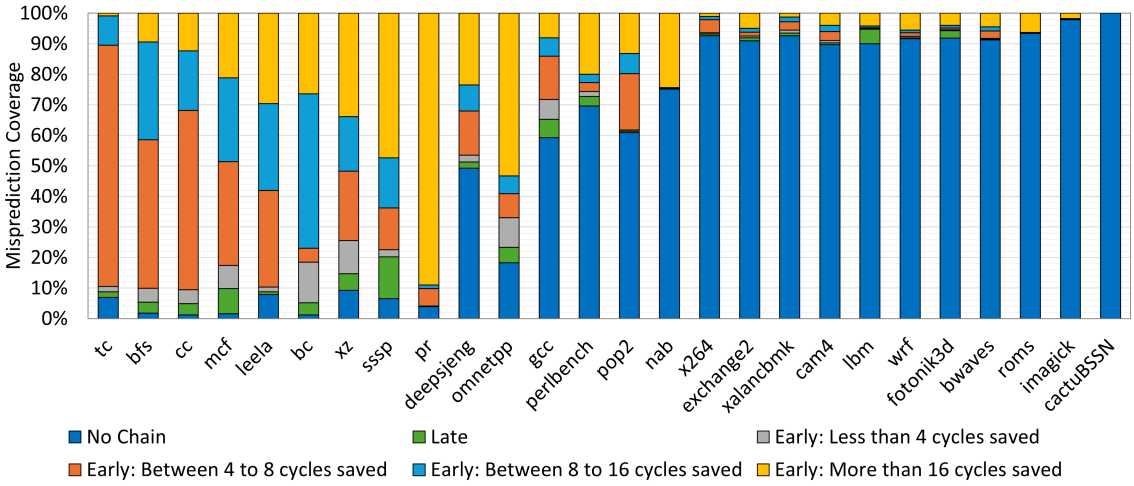

Figure 4.7: Branch misprediction coverage

Benchmarks	tc	bfs	cc	mcf	leela	bc	xz	sssp	pr
Full penalty saved	0.1%	11.05%	11.12%	2.23%	4.08%	16.46%	16.71%	19.27%	90.45%
Benchmarks	deepsjeng	omnetpp	gcc	perlbench	pop2	nab	x264	exchange2	xalancbmk
Full penalty saved	12.8%	44.11%	2.32%	15.87%	5.85%	11.82%	0.61%	3.74%	0.61%
Benchmarks	cam4	lbm	wrf	fotonik3d	bwaves	rooms	imagick	cactuBSSSN	
Full penalty saved	3.61%	0.83%	1.43%	6.16%	3.3%	5.9%	1.61%	0%	

Table 4.2: Percentage of branch mispredictions for which the full penalty is saved

chains (reducing the average load latency for some benchmarks) and provides some of the multiplicative benefits discussed in Section 1.1.

The precomputation thread density varies widely between applications, as seen in Figure 4.6.c. The precomputation thread reduces the number of main thread instructions fetched well below the baseline as it significantly cuts out on wrong-path instructions. In *pr*, this reduction is so large that even with the precomputation thread, the total number of instructions fetched is slightly lower than the baseline.

Coverage: Figure 4.7 shows the misprediction coverage of the CDE precomputation thread. Among branch-intensive benchmarks (>1 MPKI), over 75% of mispredicted branches show some reduction in misprediction penalty, covered under the "Early" categories. Benchmarks with very high MPKI (>10) mainly show a 4-8 cycle reduction since mispredictions tend to be clustered in these benchmarks.

The precomputation thread can only run ahead a short distance before it detects a misprediction, which synchronizes the state of both threads. In contrast, the distance between successive mispredictions is larger in *sssp*, *pr*, *omnetpp*, *perlbench*, and *nab*. The precomputation thread has more opportunities to run ahead as it skips more instructions in these benchmarks. Table 4.2 shows the percentage of mispredictions for which the full penalty is saved.

Late Precomputations: *Mcf, sssp,* and *gcc* have a non-trivial percentage of "late" precomputations. This predominantly occurs when the precomputation thread and main thread branches finish execution in the same cycle. There are very few cases (<0.001 PKI) where the main thread branch resolves first.

If only late precomputations are seen in the last 100K instruction window, precomputation is terminated for 1M instructions. Without these terminations, performance drops from 4.3% to 4.1%. The biggest difference is seen in *nab, x264, exchange2,* and *bwaves* - they show a performance drop of 1-2% due to backend contention. I experimented with measurement windows of 10K to 10M instructions. Windows smaller than 100K do not capture enough late precomputations. Any window higher than 100K instructions works, but a 100K measurement window followed by a 1M runtime window ensures any spurious terminations don't hurt performance.

These terminations are prominent in low-branch MPKI benchmarks and are responsible for most mispredictions not covered by the precomputation thread - the "No Chain" category in Figure 4.7. Only $\sim$ 5% of the mispredictions under this category are not covered due to incorrect chains and Block Cache misses.

4.6.3 Load Prefetching Effect

Run-Ahead Distance: Figure 4.8 shows a breakdown of the run-ahead distance for precomputation thread instructions. For example, in *bfs* 70% of precomputation thread instructions are fetched 1 to 8 cycles ahead of their main thread counterparts, and the remaining 30% are fetched 8 to 64 cycles ahead. The clustered

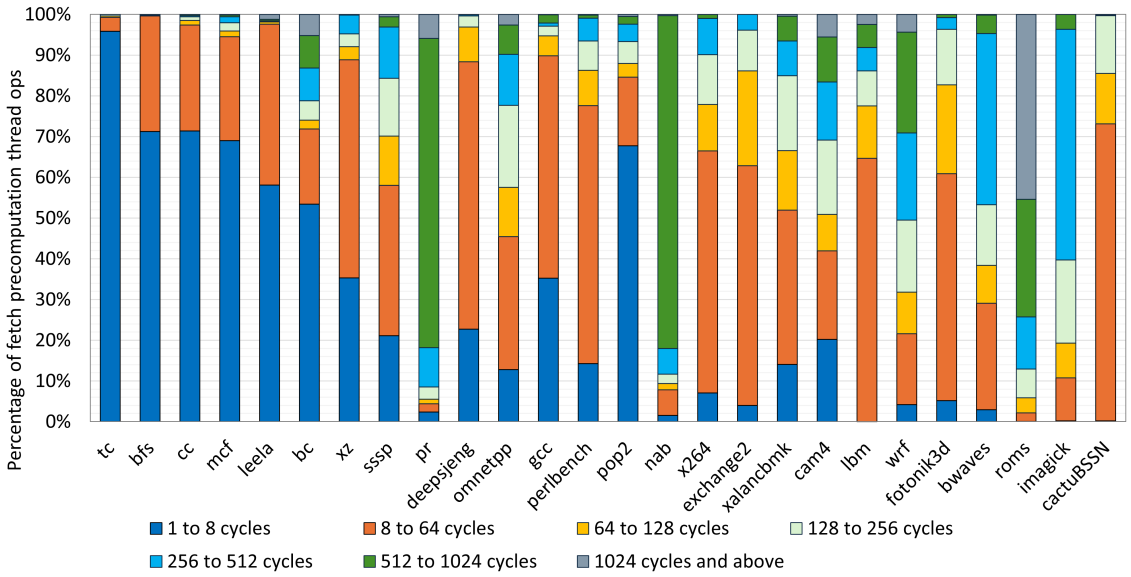

Figure 4.8: Run-ahead distance

mispredictions in high-branch MPKI benchmarks are reflected in their low ahead distance as the precomputation thread state is frequently synchronized with the main thread after mispredictions in these benchmarks.

The ahead distances suggest that the precomputation thread may be able to prefetch loads that hit in the Mid-Level or L2 Cache (MLC) and the Last-Level Cache (LLC) in many applications. If the precomputation thread runs far enough ahead, it can issue a load many cycles before its main thread counterpart. This precomputation thread load effectively acts like a prefetch instruction, bringing in data from higher-level caches into the D-Cache. Thus, when the main thread load is eventually executed, it hits in the D-Cache. Benchmarks like *pr*, *nab*, and *roms* can potentially hide the full memory access latency as they can fetch precomputation thread instructions more than 500 cycles ahead of the main thread on average.

To measure how much of the benefit comes from prefetching loads, Figure 4.9 shows the number of main thread loads that missed in the D-Cache and went out to memory, hit in the LLC, or hit in the MLC (with the CDE precomputation thread

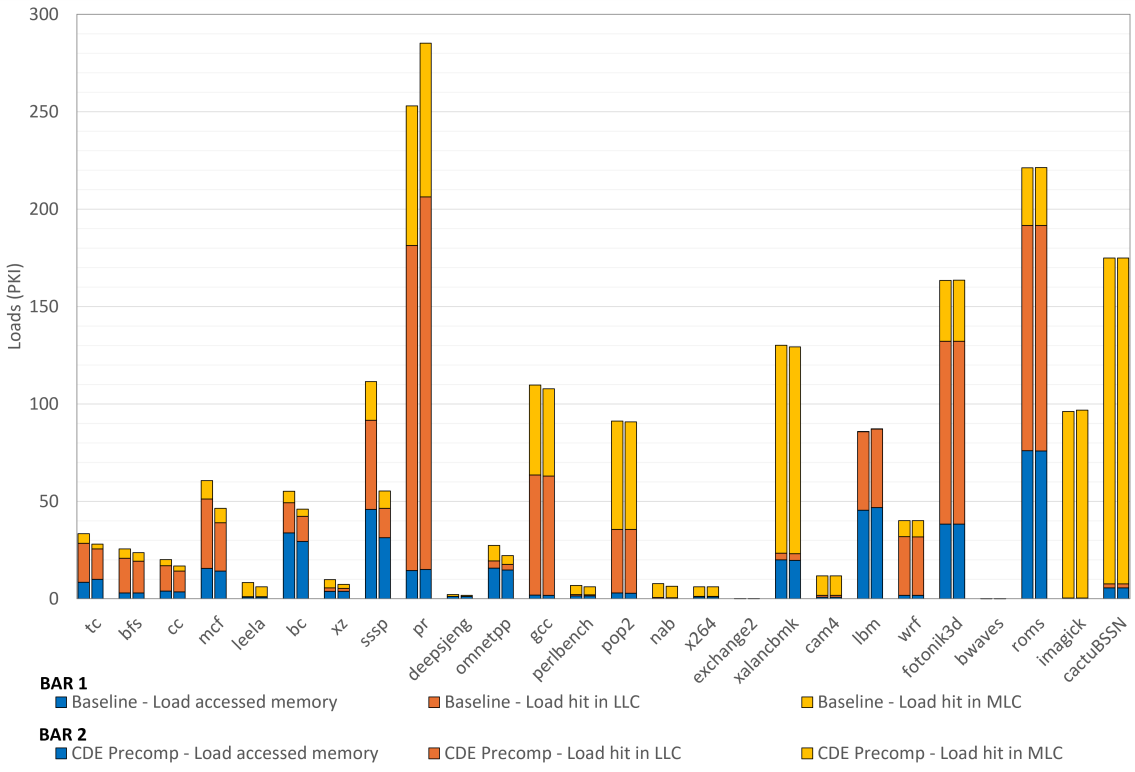

Figure 4.9: Distribution of main thread load accesses that miss in the D-Cache

running alongside it). The lower yellow and orange bars show that loads in hard-to-predict branch chains that otherwise access the MLC or the LLC are converted into D-Cache hits by the precomputation thread in several benchmarks, especially if the ahead distance for these benchmarks is high (*bc, sssp*). However, there are exceptions like *pr*, where precomputation thread loads are either prefetched too early or cause a lot of backend contention, suggesting that secondary effects limit the benefit of the load prefetching.

The actual performance impact of these prefetches is minimal - turning off early resolution in the precomputation thread (in which case any benefit it provides is due to prefetching) reduces the performance gain to 0 in all benchmarks. The bulk of the performance benefit associated with prefetching loads comes from hiding larger latencies (i.e., reducing the blue bar depicting memory accesses), which requires

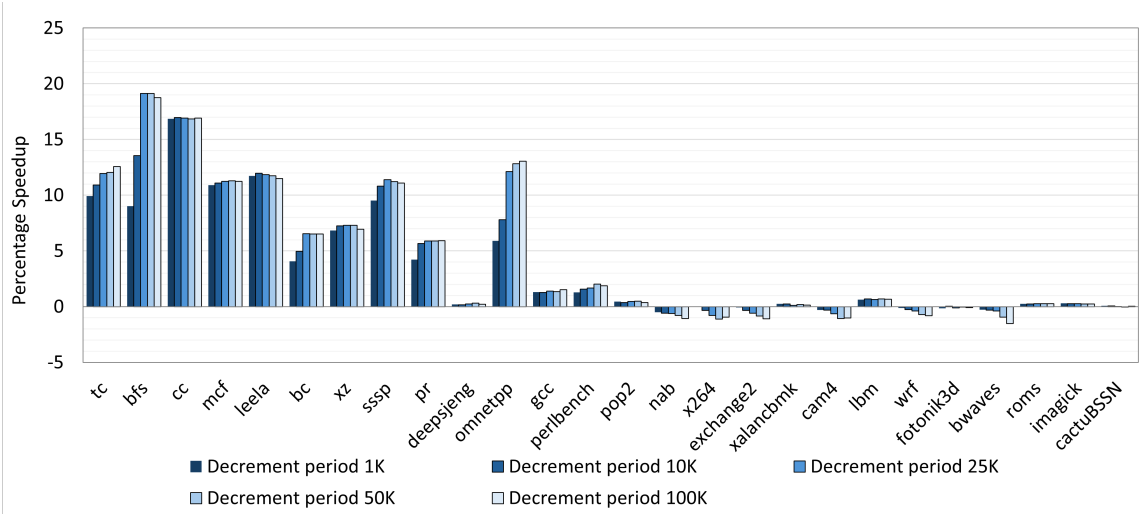

Figure 4.10: Performance for different Critical-Branch Count Table decrement periods

large ahead distances. However, in benchmarks with a high branch MPKI, larger ahead distances can only be achieved if intermediate mispredictions are also resolved. Moreover, saving MLC and LLC hit latencies only provides noticeable benefits when branch mispredictions are also simultaneously improved.

Only using load chains in the precomputation thread, even for long-latency loads specifically, is thus unlikely to provide much performance. This is evaluated in Chapter 6, and only provides a 2.2% geomean performance gain.

4.6.4 Varying the Precomputation Thread Density

Figure 4.10 shows how the speedup changes with the number of hard-to-predict branch chains in the precomputation thread. Varying the decrement period changes the MPKI threshold at which branches are marked hard-to-predict. A lower decrement period only marks fewer, higher-MPKI branches, while a higher decrement period provides better coverage. Overall, while most benchmarks benefit from a higher misprediction coverage, there is a sweet spot that balances out coverage vs timeliness, apparent in *leela* and *bfs*.

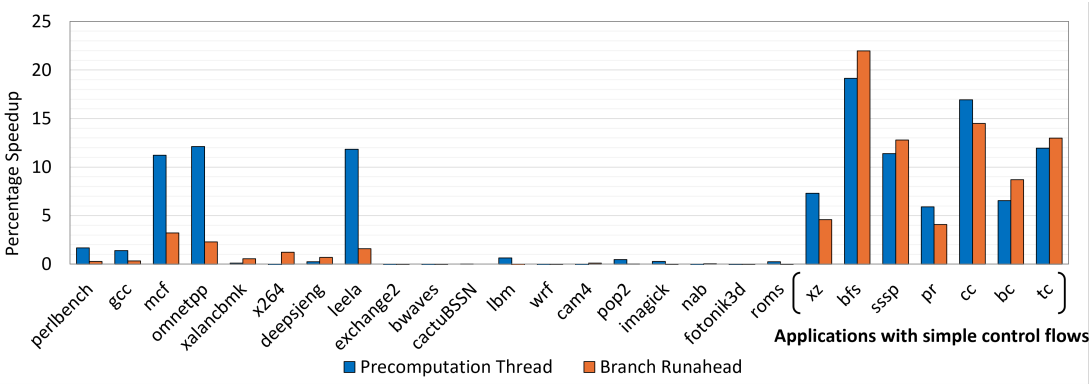

Figure 4.11: Comparison against Branch Runahead

4.6.5 Comparison against Branch Runahead

I implemented a scaled-up version of Branch Runahead using a dedicated execution engine with the same baseline OoO core and measured its performance improvement. This is shown in Figure 4.11.

The CDE precomputation thread overall does better than Branch Runahead (geomean 3.1%) even though it uses on-core execution resources compared to Branch Runahead's dedicated execution engine (which has a large backend optimized to execute branch dependence chains). However, Branch Runahead performs comparably on benchmarks with simple control flows. This includes all the GAP benchmarks and *xz* from SPEC. Branch Runahead explicitly identifies independent branches within a loop using merge point prediction in such applications and executes multiple instances of their dependence chain in parallel. The precomputation thread saves fewer cycles per-branch in simple control flow applications but makes up for it through higher misprediction coverage (∼90% vs ∼30% on these benchmarks). Branch Runahead does better than the CDE precomputation thread on *bfs*, *sssp*, *bc*, and *cc* primarily because it has a lot more backend execution resources available.

Benchmarks with more complex control flows have fewer independent branches. Branch Runahead performs poorly here as it is harder to extract misprediction level parallelism or leverage control independence in these benchmarks.

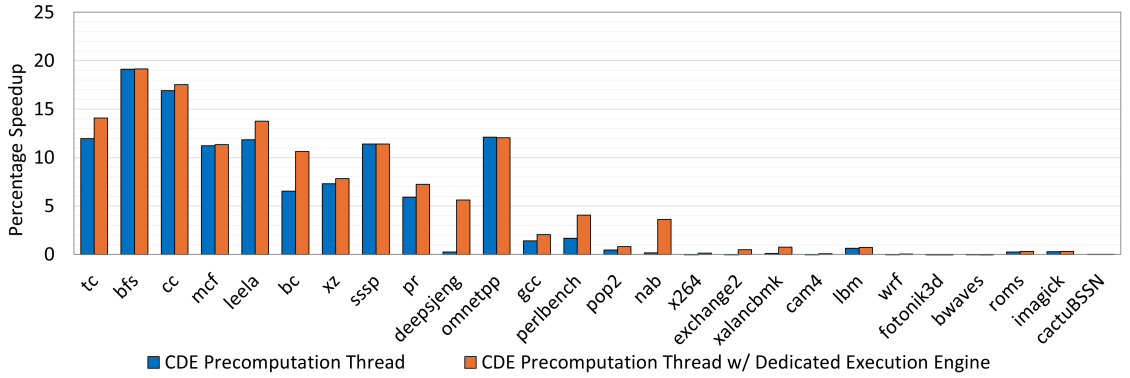

Figure 4.12: Speedup with a dedicated execution engine

4.6.6 On-Core vs Dedicated Execution Engine

Executing the precomputation thread on a separate execution engine eliminates any interference associated with using on-core resources. Figure 4.12 shows the performance of the precomputation thread when run on a dedicated execution engine with the same amount of resources as the core backend (400 Physical Registers, 256 Reservation Stations, and 16 Functional Units). Both threads still share the same D-cache ports and MSHRs. This configuration pushes the overall performance up to 5.3%. This is not a significant increase, given the execution engine has a much larger area and power overhead compared to using on-core resources. Most of the improvement comes from *bc*, *deepsjeng*, *perlbench*, and *nab* - these mainly benefit from the reduced contention for Functional Units.

4.6.7 More Sensitivity Studies

Impact of mask cache on chain accuracy: Figure 4.13 shows the PKI of incorrect chains detected (due to incorrect register or memory dependencies), with and without the mask cache. Keeping this number below 1-2 PKI ensures the precomputation thread isn't terminated frequently and can run ahead as far as possible. Without the mask-cache, the high-branch MPKI benchmarks chains are terminated much more frequently, as only the most recent version of the dependence chains is

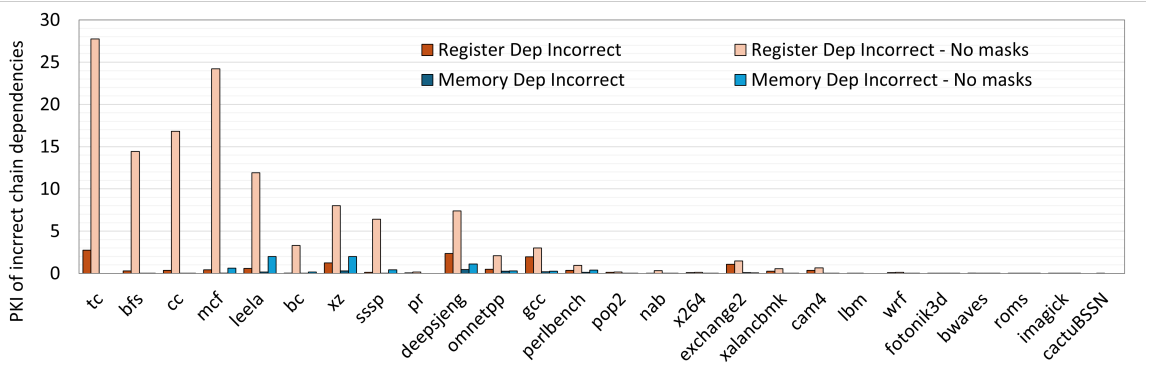

Figure 4.13: Incorrect chains detected per 1000 instructions

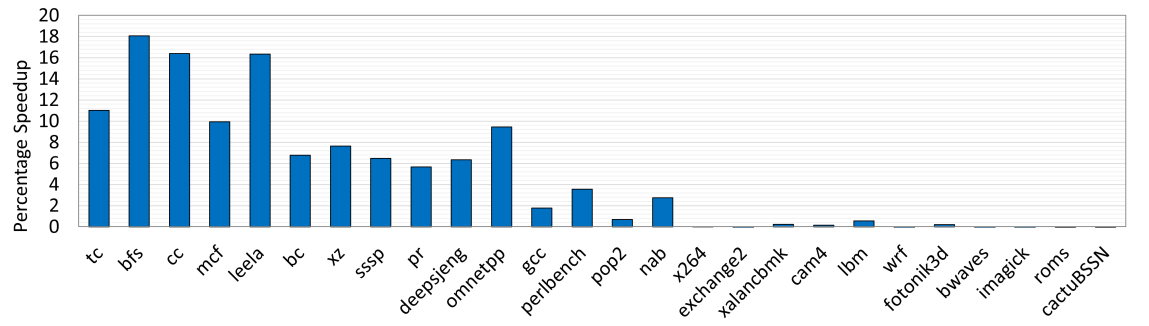

Figure 4.14: Relative Speedup with 32 Functional Units in the Baseline

recorded. In terms of performance, this reduces the overall geoeman gain to just 2%, with most of the high-branch MPKI benchmark suffering the most. A small handful of benchmarks see a slightly larger run-ahead distance without the mask cache. However, adding more instructions to the chains to ensure they are correct across all previously seen control flows always improves overall performance.

Relaxing Functional Unit Constraints: Adding more Functional Units to the baseline (32 universal execution ports) increases the relative speedup for the CDE precomputation thread to only 4.6% (note that the baseline for this bar is different). Figure 4.14 shows that the current implementation already utilizes functional units efficiently, and only specific benchmarks like *leela*, *deepsjeng*, and *nab* show benefit.

Chapter 5: Preferential Allocation for Long-Latency Load Chains

Criticality Driven Execution expands the instruction window for long-latency loads to improve Memory Level Parallelism (MLP). This is achieved by prioritizing fetch and allocation for long-latency load chains and reducing the proportion of backend window resources allocated to the remaining "non-critical" instructions.

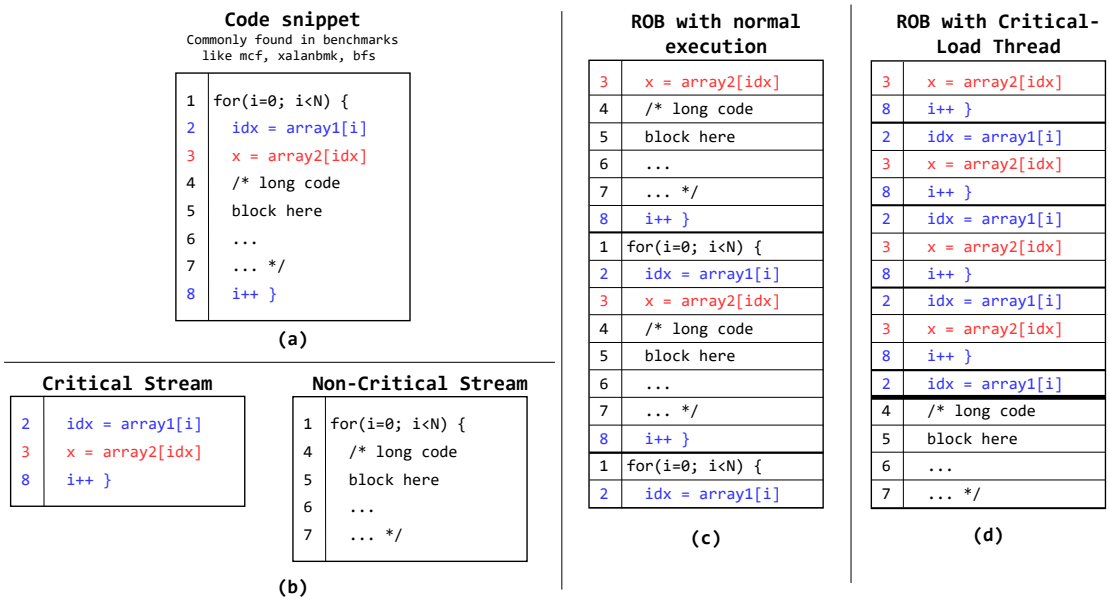

Figure 5.1: Preferential allocation example

5.1 Improving Memory Level Parallelism

The example in Figure 5.1 depicts how this works. The load on line 3 (Figure 5.1.a) is a long-latency load; when executed normally, two iterations of the code block and thus two instances of the long-latency load can fit in the Reorder Buffer (ROB) simultaneously (Figure 5.1.c). CDE first breaks this code into two instruction streams: the critical stream comprising long-latency load chains, and the rest of the

instructions in the non-critical stream (Figure 5.1.b). Instructions in both streams are part of the main thread and commit their results. Under preferential allocation, the critical stream is fetched and allocated first, and non-critical instructions are initially skipped. Backend window resources - the Re-order Buffer, Reservation Stations, Physical Registers, and Load and Store Queues - are partitioned to provide more entries to the critical stream. This allows critical instructions to span a sequential instruction window larger than the ROB size. Figure 5.1.d depicts the partitioning using a solid line with 80% of ROB entries assigned to the critical stream. Consequently, four dynamic instances of the long-latency load reside in the ROB simultaneously, providing higher MLP than the ROB under regular execution.

Non-critical instructions are fetched later into the smaller ROB partition. Instructions are allocated in program order within their respective partitions. In-order retirement is maintained by comparing the oldest instructions in each partition, facilitated by the timestamps assigned to the chain instructions in the critical stream. When instructions in either section retire, more are fetched to fill the corresponding ROB partition.

5.1.1 Partitioning Backend Resources

Preferential allocation improves parallelism for long-latency load chains by lowering non-critical instruction throughput as they are fetched later and get fewer ROB entries. Non-critical instructions often comprise simple arithmetic operations or stores that can be executed quickly. They are not impacted much by this reduced throughput and do not block the retirement of critical instructions, which normally have a much higher latency. However, there may be some execution phases where non-critical instructions require more resources. This happens when they contain many floating-point operations or loads that hit in the LLC. To balance the throughput of both streams, CDE has a dynamic partitioning algorithm that uses the relative number of full window stalls in each partition to adjust their sizes.

5.1.2 Impact on Branch Misprediction Latency

Hard-to-predict branches are de-prioritized in preferential allocation, which increases their resolution latency in some benchmarks. Adding hard-to-predict branch chains to the critical stream helps alleviate this effect and claws into some of the multiplicative benefits of accelerating both chains (improving the speedup for preferential allocation to 6%). This benefit is limited as preferential allocation cannot fetch and execute branch chains as fast as the CDE precomputation thread. However, one key advantage of preferential allocation is that it does not increase backend contention. It can thus provide higher coverage by accelerating chains simultaneously without slowing down non-critical instructions, a property used in the unified model in Chapter 6.

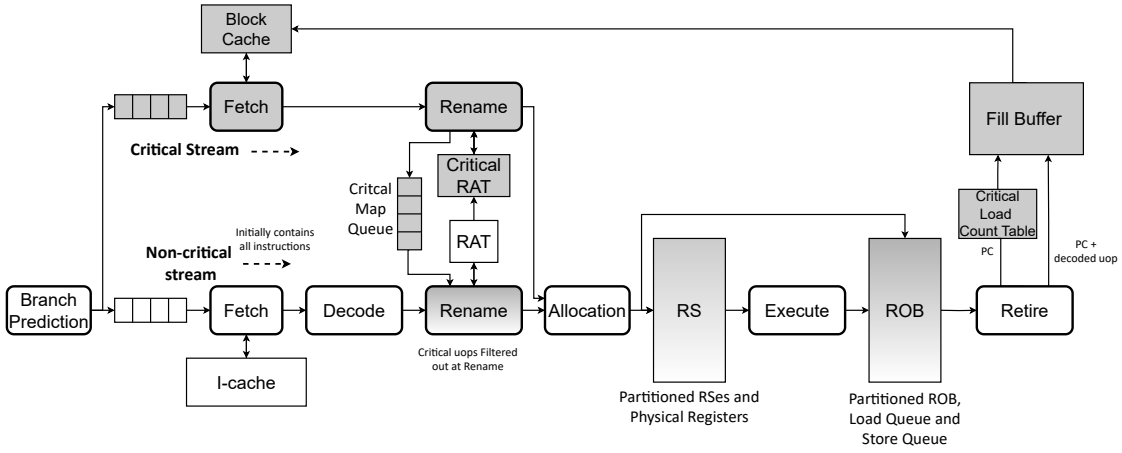

Figure 5.2: Implementation overview: Preferential allocation for load-latency loads

An overview of the changes needed to support preferential allocation is shown in Figure 5.2. The Critical Count Tables and Fill Buffer work as described in Chapter 3. The critical stream is constructed by reading out Block Cache entries via the shadow Fetch stage, similar to the CDE precomputation thread (Section 4.3.1). The fetched critical stream instructions are sent to an 8-wide shadow Rename stage with a critical RAT that tracks dependencies for the critical stream. One key additional

structure is the Critical Map Queue, which records the Physical Registers assigned to the critical stream. Its instructions are then sent to the allocation logic.

In parallel, instructions are fetched from the I-cache, forming the non-critical stream, which initially contains all instructions. After Rename, critical uops are filtered out, and only non-critical uops are forwarded. This ensures no duplicate uops enter the backend. Window resources are partitioned into two sections. The partition sizes are dynamic but skewed towards a larger critical section. Instructions are filled into their respective ROB partitions in program order, and the oldest instructions in each partition are compared for in-order retirement. The structure parameters are summarized in Table 5.1.

Core	Critical Stream FE: dedicated 8-wide Fetch and Rename Stages, 9-cycle latency Allocation-ports shared with the main thread 75% ROB, RS, PR, LQ, and SQ entries assigned to critical stream initially, dynamically varied
Caches	Critical-Load Count Table: 256-entry cache, 0.2KB, 1-cycle access Block Cache: 512-entry cache, 256 zero-tags, 8-way, 19KB
Other structures	Fill Buffer: 512-entry queue, 8KB, single access port Critical Map Queue: 0.8KB, critical RAT

Table 5.1: Structure sizes for the preferential allocation model

5.2 Frontend

5.2.1 Fetch

The Fetch unit and Block Cache design for preferential allocation is the same as the CDE precomputation thread, except the Block Cache in this implementation holds long-latency load chains (and hard-to-predict branch chains depending on the configuration). Preferential allocation is initiated on a hit in the Block Cache.

5.2.2 Rename

Contents of the main RAT are copied into the critical RAT when preferential allocation is initiated. Since critical uops consist of dependence chains, they can

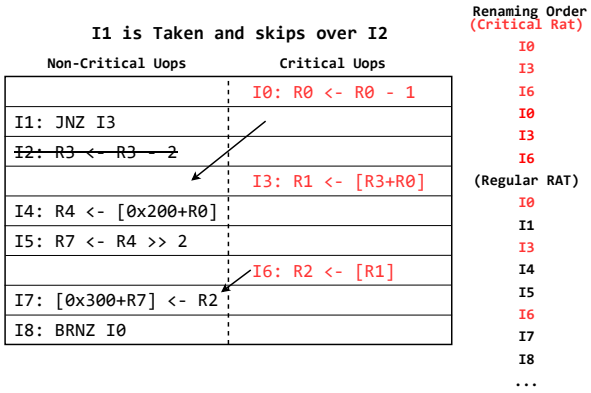

Figure 5.3: Renaming in preferential allocation: An example

be correctly renamed using just the critical RAT. All Physical Registers assigned to critical uops are recorded in the Critical Map Queue, a 368-entry FIFO. The main Rename stage processes the non-critical stream, which initially contains both critical and non-critical uops.

The critical stream does not use results produced by non-critical uops[1]; any uop in the dependence chains of a critical uop will also be marked critical in the Fill Buffer when tracing chains. However, non-critical uops can use results produced by critical uops. Consider the example in Figure 5.3. The critical stream is highlighted in red. $I4$ and $I7$ are non-critical but depend on results produced by critical uops.

To ensure these instructions read the correct data, the Renaming logic for the non-critical stream is modified to read from the Critical Map Queue and the Free List. Non-critical uops in this stream are renamed normally using the Free List, but critical uops use the Physical Registers saved in the Critical Map Queue to update the main RAT. This ensures the state of the main RAT is updated in-order and non-critical uops that depend on the critical stream read the correct Physical

[1]The critical thread uses the results produced by older uops before preferential allocation is initiated - these dependencies are set up via the initial RAT copy.

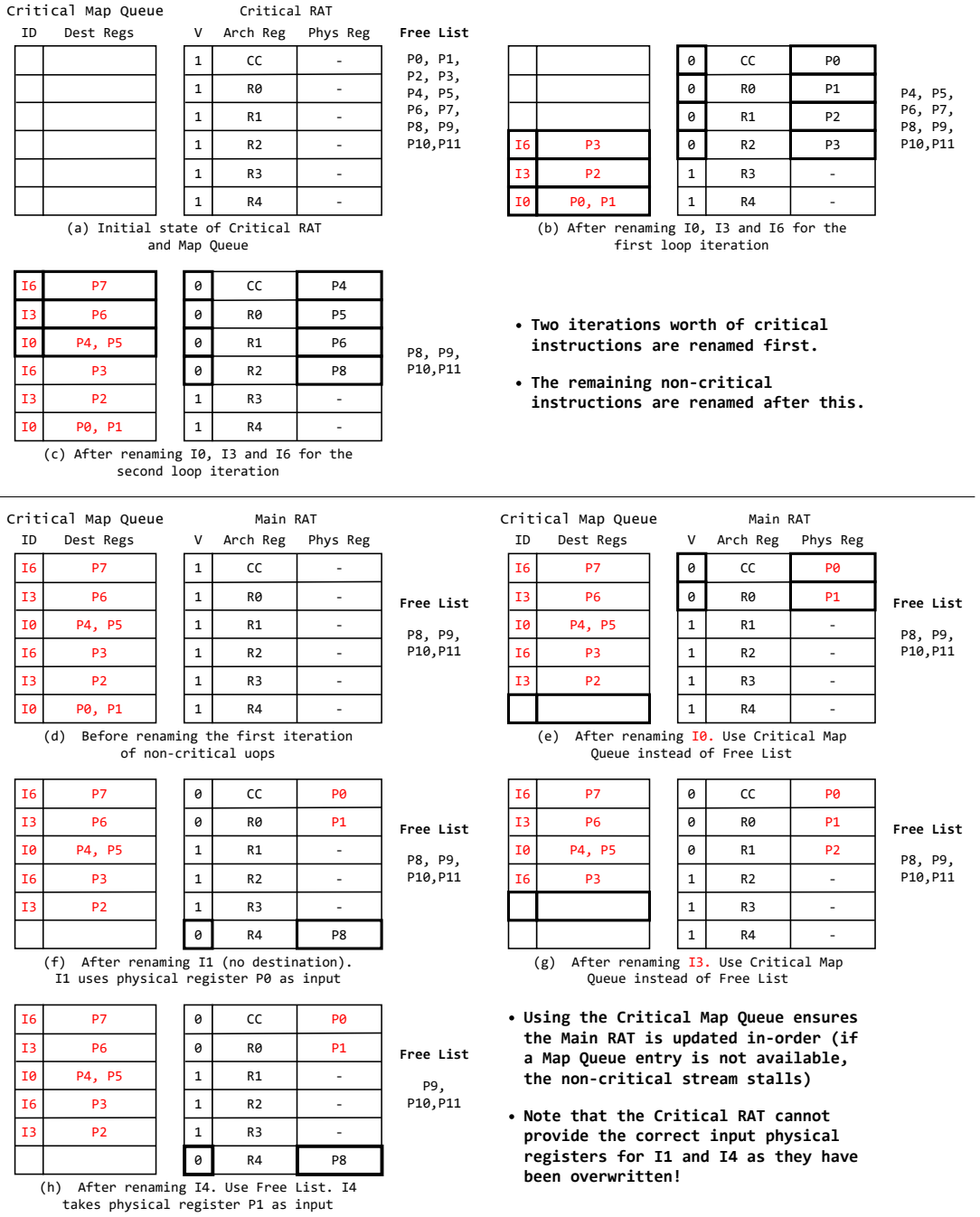

Figure 5.4: Renaming in preferential allocation: An example (continued)

Registers. Figure 5.4 shows this process for the example in Figure 5.3. From the regular Rename stage, only non-critical uops are allocated to the processor backend, ensuring no duplicate critical uops enter the backend.

The intervening operations on the Critical Map Queue prevent non-critical uops from being renamed at peak bandwidth. However, this does not degrade overall performance as non-critical instructions can tolerate the reduced bandwidth. A fail-safe mechanism using one poison bit per RAT entry prevents incorrect execution, as discussed in the following section.

5.2.3 Dependence Violations in the Critical Stream

The dependence chains constructed as part of the Backward Dataflow Walk can be incorrect in rare cases. This primarily happens when a register dependency spans over 512 uops (the capacity of the Fill Buffer) or the current control has not been seen in the Fill Buffer (for instance, if branch I_1 in Figure 5.1 is not-taken). A register dependence violation occurs when a critical uop that depends on an incorrectly marked non-critical uop is fetched and renamed before the non-critical uop, leading to incorrect execution. While these violations are rare ($<$ 0.01 PKI), they need to be detected and resolved to ensure correct execution.

The register poisoning scheme discussed in Section 4.4.4 is used to detect register dependence violations. Since the non-critical stream Renames both critical and non-critical uops, the poisoning scheme works as-is[2]. Memory dependence violations are detected and resolved by the memory disambiguation logic discussed in Section 5.3.4. On a dependence violation, all instructions younger than the offending critical uop are flushed, and regular execution resumes at the next instruction. Since these flushes are rare ($<$0.01 PKI), their impact on performance is minimal.

[2]Instruction I_3 in both examples (Figure 4.5 and Figure 5.3) causes a violation

5.3 Backend

5.3.1 Scheduling

Oldest-first scheduling decides which Reservation Stations uops are sent to the Execution Units. No additional allocation or scheduling bandwidth is added. Since critical uops enter the Reservation Stations first, they are naturally prioritized with oldest-first scheduling.

5.3.2 Dynamically Changing the Partition Sizes

A smaller partition size for the non-critical thread reduces its throughput but does not generally affect performance. However, assigning too small a partition will eventually lead to slowdowns. This effect is more pronounced when long-latency loads in the critical instruction stream hit in the cache hierarchy (not all dynamic instances of loads marked by the Critical Count Tables cause LLC misses).

A dynamic partition algorithm allows the partition sizes to vary by maintaining two sets of fill and retire pointers for the ROB, Load Queue (LQ), and Store queue (SQ). Their increment/decrement logic is modified to use a register that keeps track of the boundary between the two partitions[3]. The partition sizes are adjusted by changing the entry that the boundary register points to.

The ROB, LQ, and SQ partitions are independently varied. The partitioning mechanism is controlled by counters that measure the number of full window stalls for both streams. If the counter value for one stream exceeds the other, its partition size is increased and the counters are reset. ROB partition sizes are incremented/decremented by 32 entries. The LQ/SQ partitions use a granularity of 8. Reservations Station and Physical Register are allocated on a first-come-first-served basis, with 32 entries reserved for each stream to prevent deadlocks.

[3]Given the ROB, LQ, and SQ are treated as circular buffers

5.3.3 Branch Mispredictions

Preferential allocation deals with branch mispredictions in the same way as a regular OoO core. On a misprediction, all instructions younger than the mispredicted branch are flushed. The Critical Map Queue is partially flushed; this does not impact the misprediction latency as entries in the Critical Map Queue are filled in-order.

The state of the critical RAT is checkpointed and recovered using the same mechanism as the main RAT. A misprediction flush does not terminate preferential allocation if there is a Block Cache entry for the next fetch address after the misprediction.

5.3.4 Consistency Considerations and Memory Disambiguation

In OoO cores with Total Store Ordering (TSO) or more relaxed forms of memory ordering, associative lookups are performed on memory addresses and instruction timestamps in the Load and Store Queues to ensure memory operations are executed correctly. Since timestamps are available for both critical and non-critical uops, the memory disambiguation logic is relatively unchanged. The number of timestamp bits to be compared increases (by 1-2) as instructions under preferential allocation span a larger instruction window than the ROB.

There may be non-critical stores not allocated to the SQ that are in program order before critical loads in the LQ. However, LQ entries for critical loads are held until Retire. Since retirement occurs in-order, the missing non-critical stores will be allocated SQ entries before any younger critical loads commit. The LQ can then be checked for memory dependence violations when the store address is calculated, ensuring correctness.

5.3.5 In-Order Retirement

Instructions in the critical and non-critical sections of the ROB are each filled in program order. Thus, only the oldest instructions in each section - as indicated

by the two retire pointers - need to be checked for retirement. A simple comparison of their timestamps indicates the next instruction to be retired. While this increases the complexity of the retirement logic, it does not affect performance noticeably as very few programs are limited by Retire stage latency.

5.3.6 Terminating Preferential Allocation

Preferential allocation is terminated if the fetch unit encounters a miss in the Block Cache or a dependence violation is detected. Following this, any remaining non-critical instructions corresponding to fetch addresses buffered in the main Fetch Queue are fetched. Once the last non-critical instruction is fetched, the processor resumes regular execution. Since the non-critical stream renames all instructions, the state of the main RAT is the same as if all instructions were renamed in program order. No additional work needs to be done to transition back to regular execution. When the frontend stops fetching critical uops, there may be un-retired critical instructions in the pipeline. To deal with this, all instructions fetched regularly are treated as non-critical, and the critical partitions for all backend structures are gradually decreased.

5.4 Hardware Overhead

The area and power overhead for the preferential allocation model and the CDE pre-computation thread are the same since both models use the same high-level structures and are sized similarly.

Energy: The energy overhead of all additional structures adds up to 1.4% - the Block Cache and critical RAT contribute to most of this. The added FIFOs and pipeline logic do not have significant overhead since the read and write operations and static energy for these are much lower due to their lower complexity. Overall, the preferential allocation model provides a significant 5.1% reduction in energy consumption (with select branch chains included) as it reduces execution time without increasing the number of dynamic instructions that enter the backend.

5.5 Evaluation

Methodology: The same simulation infrastructure and benchmarks outlined in Section 4.6 are used to evaluate the preferential allocation model.

5.5.1 Performance

Figure 5.5.b shows the speedup of preferential allocation over the baseline OoO core for two configurations. The first only accelerates long-latency load chains, providing a geomean improvement of 3.1%. The second has a few additional branch chains. It uses the Critical-Branch Count Table with a decrement period of 500 (alongside the Critical-Load Count Table) and reaches a geomean improvement of 6%. The benchmarks are sorted in decreasing order of LLC MPKI (shown in Figure 5.5.a), and Figure 5.5.c shows the proportion of critical stream instructions in both configurations.

The preferential allocation model improves performance significantly when the critical stream is sparse. This allows long-latency loads to be packed together, especially in *lbm, roms, xalanbmk*, and *cactuBSSN*. Comparatively, *tc, perlbench, deepsjeng* and *nab* have fewer but more spaced apart LLC misses that preferential allocation can capture. The burstiness of LLC misses (and the corresponding split between critical and non-critical streams) has a major impact on performance. This, combined with how well the dynamic partitioning algorithm allocates resources, determines the overall performance. Average statistics on chain density and MLP do not capture these effects and thus do not correlate strongly with the observed performance improvement.

Memory Traffic: Since the critical instruction stream follows the control flow generated by the main branch predictor, it traverses the same paths as a regular OoO core. It does not increase memory traffic due to demand requests or prefetches (only the timing of the prefetches is changed).

Memory Level Parallelism Figure 5.6 shows the MLP of the two preferen-

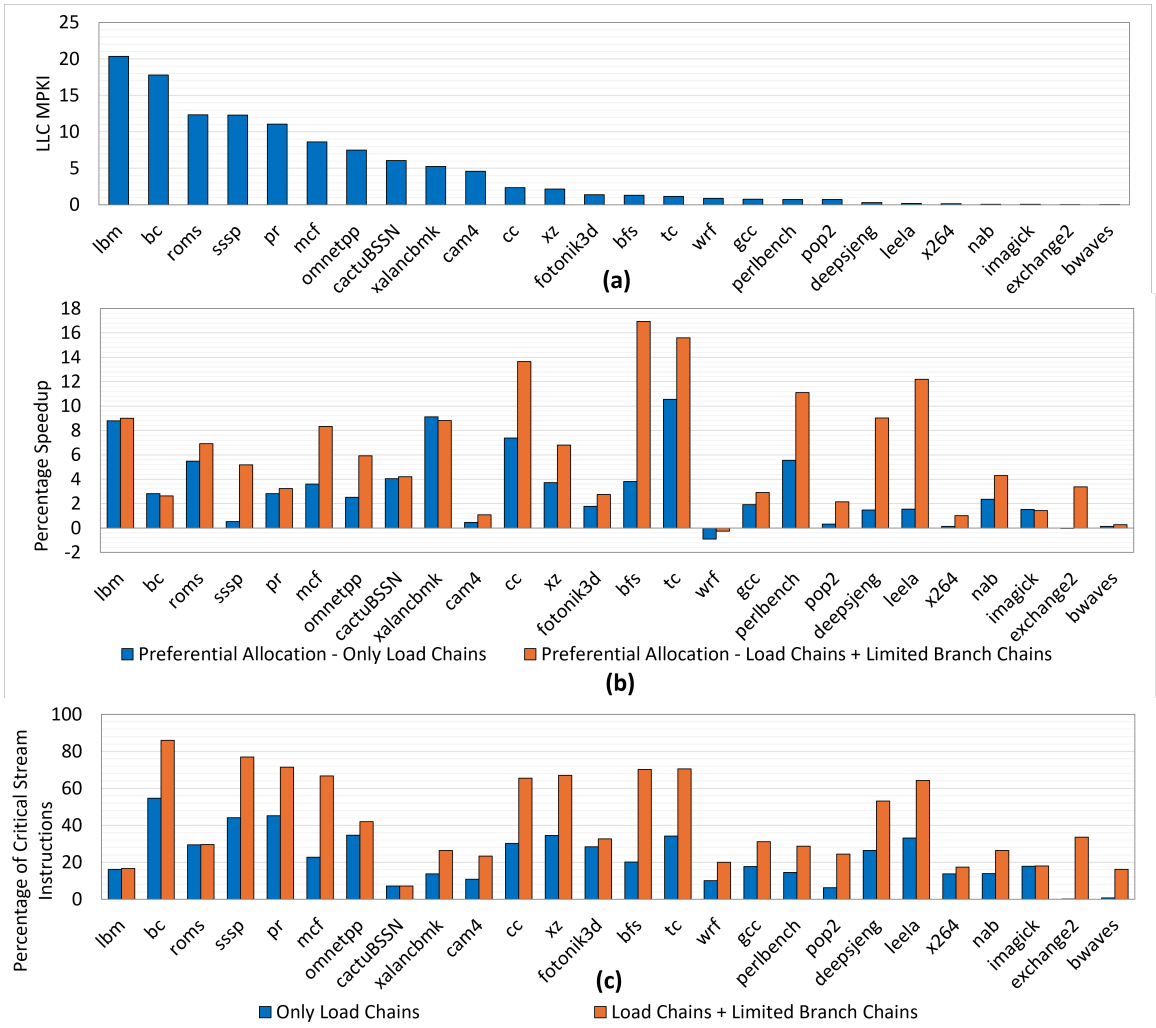

Figure 5.5: Preferential allocation: LLC MPKI, speedup, and percentage of instructions

tial allocation configurations relative to the baseline OoO core. The MLP is calculated as the average number of pending L1 misses per cycle. MLP decreases in *perlbench*, mainly due to a reduction in prefetches issued. Adding branch chains often reduces MLP as it increases the critical stream density, preventing more loads from residing in the ROB simultaneously. However, adding hard-to-predict branch chains still provides better performance, as resolving branches late in the non-critical stream is much more detrimental to performance.

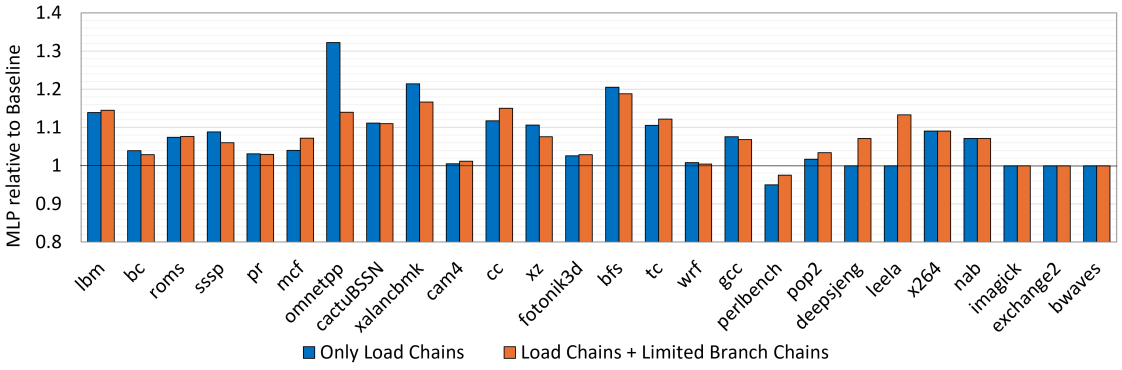

Figure 5.6: Memory-Level Parallelism

The window from which parallelism can be extracted is limited by branch mispredictions. In benchmarks with high branch MPKI and moderate to high LLC MPKI - such as *sssp, mcf, omnetpp, cc, xz, bfs, and tc* - loads beyond a mispredicted branch do not contribute to useful MLP as they are on the wrong path and generally access data not needed by the program in the near future.

5.5.2 Dealing with Branch Mispredictions

Loads beyond mispredicted branches only contribute to MLP after the misprediction is resolved. Including their chains in the critical stream reduces the adverse effects of resolving branches late as part of the non-critical stream. This provides a substantial performance improvement, as seen in the orange bar in Figure 5.5.b. Figure 5.7 shows the branch resolution latency for the two preferential allocation configurations, normalized to the baseline OoO core. Most benchmarks have a higher branch resolution latency than the baseline core for the first configuration. The second configuration reduces this effect, especially for high-branch MPKI benchmarks.

Preferential allocation resolves branches faster than the baseline in a few benchmarks. This is due to two reasons. First, improving MLP launches some long-latency loads earlier. In benchmarks where long-latency loads are in the dependence chains of hard-to-predict branches, the computation for these branches is initiated

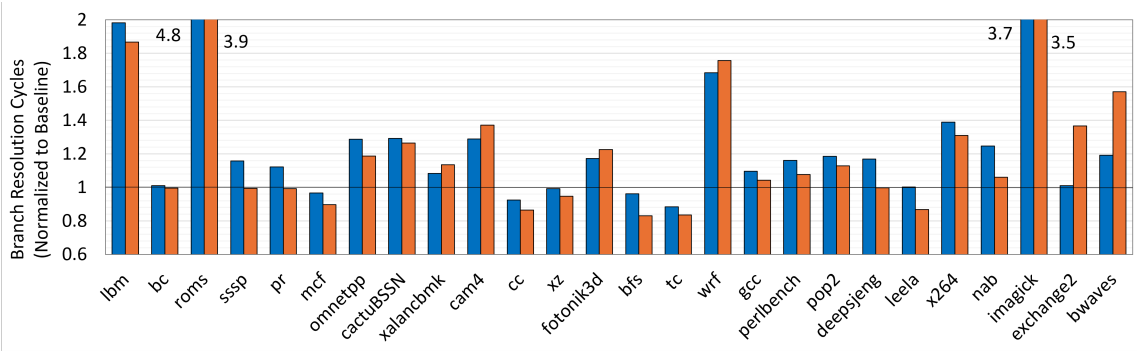

Figure 5.7: Branch resolution latency, normalized to baseline

faster. Second, the critical stream fetches fewer instructions and has a shorter frontend, reducing the effective fetch to execution latency for branches. This only saves a few cycles of misprediction penalty (∼4 cycles), but improves performance in high-branch MPKI benchmarks such as *mcf, leela, xz, cc, bfs* and *tc*.

The branch resolution latency increases in the second configuration for *cam4, fototnik3d, wrf, x264, exchange2*, and *bwaves*. These benchmarks have very few mispredictions, most of which are not covered by the critical stream as they come from infrequently mispredicted branches. However, many of their chain instructions are included in the critical stream, which delays any mispredicted branches in the non-critical stream. Since there are very few mispredictions in these benchmarks, just a few such delays end up skewing the average.

Preferential allocation can speed up hard-to-predict branch chains without executing chain instructions twice. However, it is limited by ROB size and in-order retirement, and thus cannot provide as much benefit as the CDE precomputation thread for hard-to-predict branch chains as covered in Chapter 6.

5.5.3 Varying the Critical Stream Density

Figure 5.8 shows how performance changes with the number of long-latency load chains in the critical stream for the two configurations. The configuration in Figure 5.8.a uses only load chains, while the one in Figure 5.8.b contains the additional

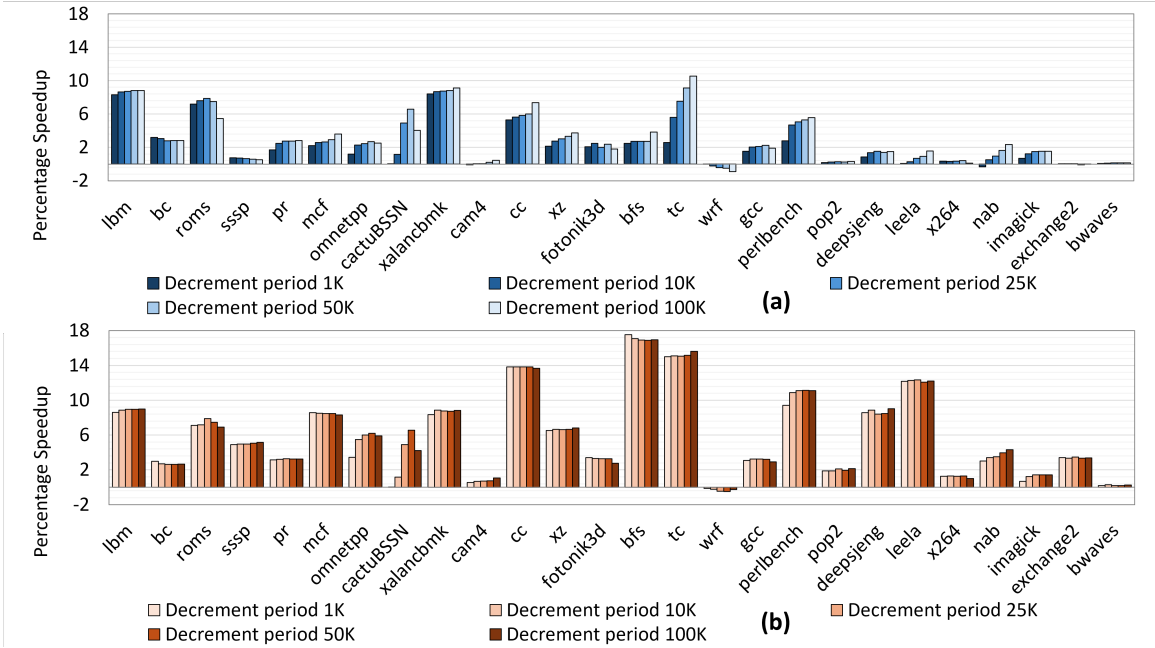

Figure 5.8: Performance for different Critical-Load Count Table decrement periods

branch chains. Most benchmarks benefit from higher coverage. *Bc, cactuBSSN, fotonik3d* and *x264* show a drop in MLP with more chains as the effect of reduced sparsity wins in these benchmarks.

5.5.4 Comparison Against Runahead Execution

Many variants of Runahead exist in academia [22, 37, 38, 39], but they share one key feature: Runahead execution is triggered on a full window stall, and the main pipeline is used to fetch and execute instructions to find more LLC misses during this full window stall.

To evaluate Runahead, I selectively initiate the critical instruction stream (without branch chains) only on full window stalls (after employing heuristics to determine when to enter Runahead mode). No start-up or wind-down costs are included, and roughly half the backend window resources are available to Runahead instructions. Figure 5.9 shows the speedup of Runahead execution compared to preferential

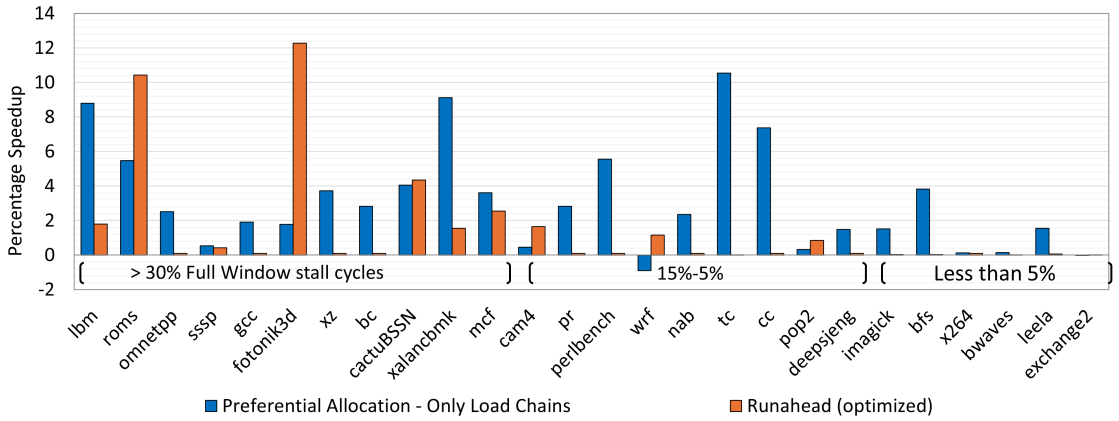

Figure 5.9: Comparison against Runahead Execution

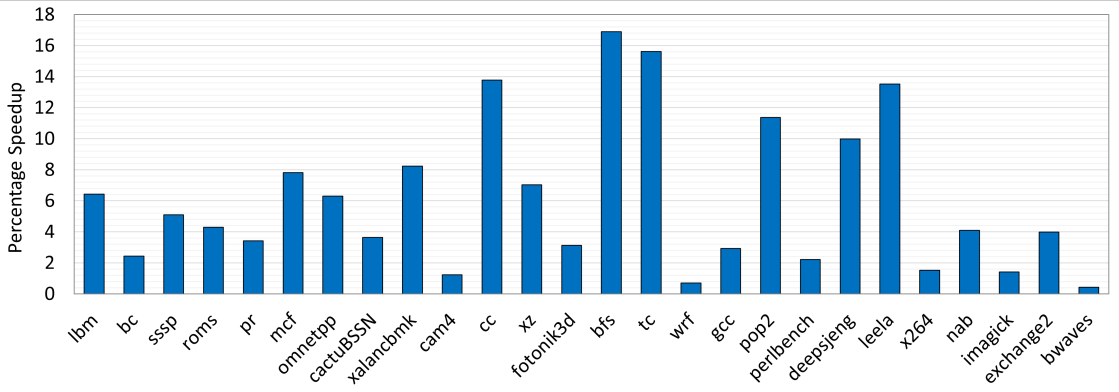

Figure 5.10: Using 24 MSHRs in the Baseline

allocation with only long-latency load chains. This only improves performance in a few benchmarks with sufficient stall cycles to sustain Runahead, as on average, only ∼7% of total execution cycles have full window stalls lasting 16 cycles or longer (Runahead cannot provide benefit for shorter stall durations). Many benchmarks show no benefit as full window stalls in a large OoO core are both short and infrequent. The Runahead does perform better in benchmarks with enough full window stalls and large runahead distances that cannot be covered by the critical instruction stream (*fotonik, roms*). However, a decoupled, speculative execution thread is better suited to extracting MLP in such benchmarks, as shown in Chapter 6.

5.5.5 Reducing the MSHR sizes

Figure 5.10 shows the speedup of preferential allocation over a baseline that contains just 24 MSHRs. The geomean speedup is 5.9% (with some branch chains included), marginally lower than the configuration with 32 MSHRs. Even with a smaller number of MSHRs, preferential allocation uses existing window resources (and MSHRs) more effectively compared to the baseline. Thus, the relative improvement does not suffer much. *lbm, mcf* and *cactuBSSN* see a drop in the speedup compared to using 32 MSHRs, as the baseline core more often saturates the MSHR capacity in these benchmarks (in which case, preferential allocation cannot improve MLP).

Chapter 6: Building a Unified Model

Understanding the properties of hard-to-predict branch chains and long-latency load chains is essential for combining their benefits. Simply accelerating both types of chains together does not necessarily improve performance, given limited on-core resources. In the CDE precomputation thread, increasing the number of chains increases backend contention and reduces the effective fetch bandwidth (Section 4.6). Preferential allocation does not increase backend contention; however, adding more chains reduces the MLP extracted in some benchmarks (Section 5.5).

This chapter builds a unified execution model that effectively combines the benefits of hard-to-predict branches and long-latency loads. To better understand how they should be accelerated, the following sections provide an implementation-independent measure of the instruction density in both types of chains and how they overlap.

6.1 Percentage of Instructions in Dependence Chains

An oracle that traces dependence chains for all branch mispredictions and LLC misses across the entire program does not reflect how actual optimizations based on these chains would work. Managing chains for individual dynamic branch and load instances is extremely storage-intensive as the number of possible chains grows exponentially with each such instance. Instead, we can capture the static branches and loads needed to achieve a specific branch misprediction and LLC miss coverage within a window. This provides a stable view of the chains within this window that can be studied. I ran the following experiment to get this data.

6.1.1 Experiment Design

First, information about instructions executed and retired in the SPEC CPU2017 and GAP benchmarks is collected. The collected instructions (in program order) are then divided into windows of 100k instructions each. Within each window, if the branch MPKI or LLC MPKI is above 1, static branches and loads that are the highest contributors to these mispredictions and misses are marked until 95% coverage is reached. These represent the hard-to-predict branches and long-latency loads within this 100k instruction window. Finally, chains are traced for all dynamic instances of the marked branches and loads within the window.

The threshold for marking branches and loads determines whether the chains focus on coverage or timeliness. Marking branches hard-to-predict using the parameters in the previous paragraph achieves an accuracy of 11% (89% of branch instances marked hard-to-predict did not cause a misprediction). In case of loads, 80% are marked long-latency but do not cause LLC misses. These incorrectly marked branches and loads increase chain density but are required to achieve high coverage.

Lowering the coverage threshold improves accuracy. Picking a threshold of 75% coverage and 5 MPKI for mispredictions/misses in the window reduces the percentage of incorrectly marked branches to 78% and loads to 69%. Chain density for these two optimization points - one targeting high coverage, and the other targeting lighter chains is summarized in Figure 6.1.

6.1.2 Benchmark Categorization

Figure 6.1 shows the percentage of instructions that are part of hard-to-predict branch chains, long-latency load chains, and part of both types of chains combined. (b) uses the high-coverage parameters, while (c) marks fewer loads and branches for better accuracy. The benchmarks are sorted in descending order of branch + LLC MPKI.

Overall, the percentage of chain instructions is lower when the focus is on

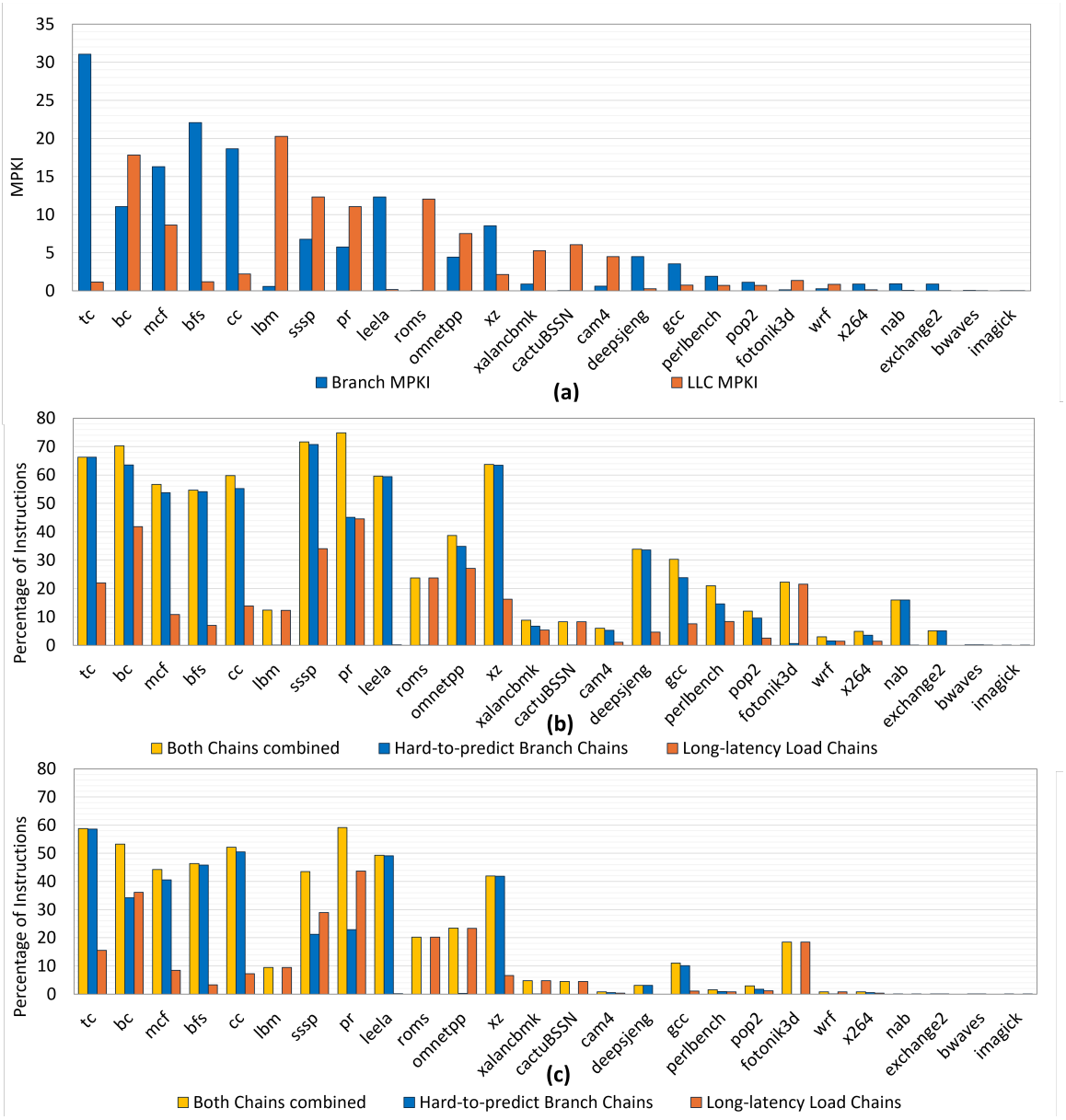

Figure 6.1: Distribution of chain instructions

accuracy (Figure 6.1.(b)). The number of hard-to-predict branch chains in particular is much lower. Fewer chains are captured for *bc*, *sssp* and *pr*, and beyond *leela*, very few branch chains are traced. Most instructions in the combined chains come from hard-to-predict branches, as seen by the slight difference between the yellow

Category	Chain Instructions	Branch MPKI	LLC MPKI	Benchmarks
I	Dense hard-to-predict branch chains, long-latency loads within these chains	Very High (> 5)	Very High (~ 10)	mcf, sssp
II	Dense hard-to-predict branch chains, long-latency loads within these chains	Very High (> 5)	Moderate (~ 1 to 2)	tc, bfs, cc, xz
III	Dense but independent hard-to-predict branch and long-latency loads chains	High (> 1)	High (> 1)	bc, pr, omnetpp, gcc, perlbench, pop2
IV	Only hard-to-predict branch chains (density varies with MPKI)	Moderate-High (0.8 to 12.3)	Very Low (< 0.1)	leela, deepsjeng, nab, exchange2, x264
V	Only long-latency load chains (density varies with MPKI)	Low (< 0.4)	Moderate-High (0.9 to 20.2)	lbm, roms, xalanbmk, wrf, cactuBSSN, cam4, fotonik
Others				bwaves, imagick

Table 6.1: Benchmark categories based on chain properties

and blue bars. Thus, adjusting the threshold at which they are marked hard-to-predict is important for regulating the percentage of chain instructions (more so than long-latency loads).

Table 6.1 categorizes these benchmarks based on how the hard-to-predict branch and long-latency load chains overlap. Most benchmarks show the same behavior under both configurations. Categories I and II contain benchmarks in which most long-latency loads are within hard-to-predict branch chains. These benchmarks are both branch and memory-intensive. The only difference is the proportion of LLC misses - the category I benchmarks, *mcf* and *sssp*, often saturate the D-cache MSHRs even under regular execution. Benchmarks in category III have relatively independent chains for hard-to-predict branches and long-latency loads[1]. Categories IV and V contain only hard-to-predict branch chains or long-latency load chains. Finally, the "Others" category represents benchmarks with a very low branch and LLC MPKI that don't fit with the rest.

Looking back at the oracle studies in Figure 1.1, category I, II, and III benchmarks benefit most from improving loads and branches together. Out of these, only accelerating hard-to-predict branch chains captures both benefits in category I and II benchmarks.

[1]A lower coverage decouples the chains in *sssp*, but the loss in coverage hurts performance significantly

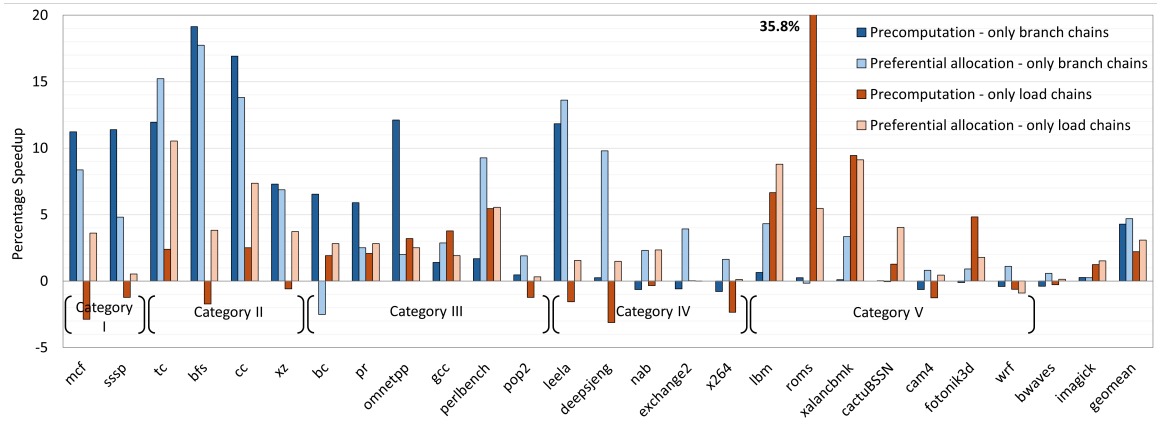

Figure 6.2: Accelerating only hard-to-predict branch chains or long-latency load chains

6.1.3 Improving Loads and Branches Individually

Figure 6.2 shows the best performing configurations for the CDE precomputation thread and preferential allocation using hard-to-predict branch chains and long-latency load chains individually.

Category I and II benchmarks do well with both execution modes when only hard-to-predict branch chains are accelerated. These chains contain many long-latency loads and thus provide the benefit of accelerating both hard-to-predict branches and long-latency loads simultaneously. Preferential allocation does better in *tc* due to less backend contention, while precomputation does better on others due to its larger run-ahead distance. These benchmarks are unlikely to show much improvement with a unified model since their hard-to-predict branch chains and long-latency load chains cannot be decoupled. Moreover, while moving a hard-to-predict branch chain to preferential allocation decreases contention, it reduces the chain's benefit. Category II benchmarks may show slightly higher improvement since their memory intensity is lower, allowing preferential allocation to better improve MLP without the intervening branch chains.

Using hard-to-predict branch chains or load-latency load chains individually

does not extract the full performance benefit in category III benchmarks. Accelerating both types of chains together is likely to show the highest benefit in these benchmarks.

Some category IV benchmarks suffer backend contention under precomputation and work better with preferential allocation. The unified model can improve on these benchmarks if the precomputation thread accelerates only a limited number of branch chains to maximize timeliness (while preferential allocation provides coverage for the rest).

Preferential allocation performs better on category V benchmarks as it can extract MLP without adding cache port pressure. However, benchmarks like *roms* and *fotonik3d* have a large run-ahead distance, which allows the precomputation thread to prefetch some LLC misses in these benchmarks.

6.2 Unified Execution Model

6.2.1 Why the Simple Approach does not Work

There are three main reasons why a smarter design is needed for building a unified model as opposed to the naive approach of simply combining the two types of chains within either execution model:

Backend contention: Contention for Functional Units, D-cache ports, and allocation bandwidth hurts performance when duplicate instructions enter the backend. Section 4.6 touches upon this - even precomputing only hard-to-predict branch chains requires early termination to avoid performance inversion in a few benchmarks. Accelerating both types of chains together only exacerbates this problem and requires actively removing some chains from the precomputation thread. However, backend contention varies widely among benchmarks and is hard to estimate without running the precomputation thread. Thus, a dynamic feedback mechanism is needed to find the sweet spot for performance.

Limited benefits with preferential allocation: While preferential allo-

cation eliminates all backend contention issues, it cannot run-ahead as far as the precomputation thread due to limitations imposed by the ROB size and in-order retirement. Figure 6.2 shows that most benchmarks with hard-to-predict branches perform significantly worse with preferential allocation than with precomputation. Thus, an independent, speculative precomputation thread is needed to accelerate some chains.

Fetch bandwidth limitations: Combining both chains reduces the effective run-ahead distance of the precomputation thread, especially in benchmarks with relatively independent hard-to-predict branch and long-latency load chains. This affects hard-to-predict branch chains predominantly as they are more sensitive to latency, and requires prioritizing their fetch and allocation over long-latency load chains.

6.2.2 Accelerating Loads and Branches together

The unified execution model for CDE addresses these limitations by both using an independent speculative precomputation thread and splitting the main thread into critical and non-critical streams. A few simple modifications to the existing mechanism for tracing chains allow CDE to dynamically manage where hard-to-predict branch chains and long-latency load chains go.

The Critical-Branch and Critical-Load Count Tables have two sets of counters instead of one (Section 6.4.1). The first set of counters marks branches and loads for the precomputation thread and has a low decrement period to maintain good timeliness and reduce backend contention. However, the load counter for the precomputation thread is **incremented only when the load is an LLC miss and the current run-ahead distance is greater than its latency**, ensuring it only captures loads that preferential allocation cannot improve. The second set of counters is used for the critical stream and has a high decrement period to provide good coverage.

Chains are traced for the precomputation thread first, and any branch or load

already marked as part of the precomputation thread chains is excluded from the critical stream. (Section 6.4.3). The decrement period for the precomputation thread branch counter is occasionally halved (every 100K instructions) to check whether reducing the number of precomputation thread instructions improves performance. Beyond a certain threshold, the precomputation thread is turned off (Section 6.5).

Finally, allocation prioritizes precomputation thread instructions over critical stream instructions, and non-critical instructions are allocated last. This, coupled with a simple first-come-first-served policy for Reservation Stations and Physical Registers, effectively manages backend resources.

These modifications allow the precomputation thread to accelerate as many chains as possible without increasing backend contention while the critical stream takes care of all the remaining hard-to-predict branches and long-latency loads.

6.3 Implementation Overview

Figure 6.3 shows an overview of the full CDE implementation that combines precomputation with preferential allocation. Instead of using three distinct frontends, this implementation removes most of the pipeline stages for the preferential allocation model. The critical stream still has a dedicated Fetch stage and associated Block Cache, but shares the rest of the pipeline stages with the non-critical stream. The precomputation thread frontend remains unchanged. Additional structures from both implementations - the Critical Map Queue, Store Data Cache, and PR Map Table - are included. Finally, the dependence chain tracing hardware (Fill Buffer and Critical Count Tables) is modified to trace two sets of chains: one for the precomputation thread and the other for the critical stream.

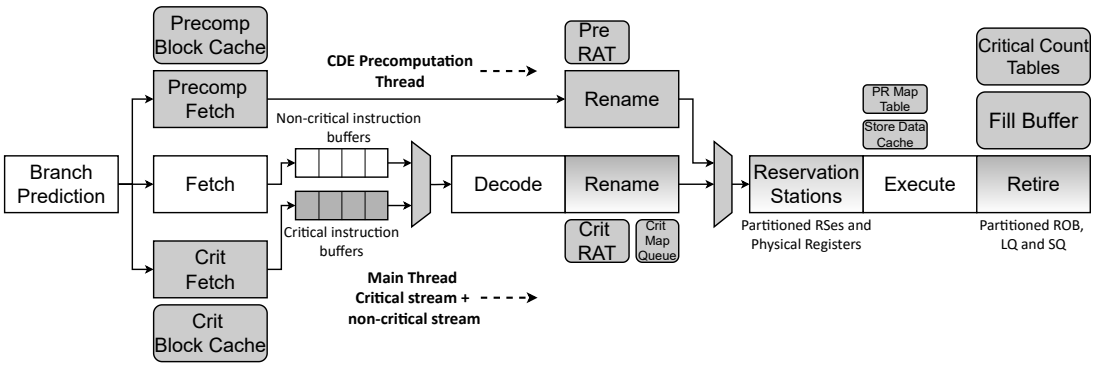

Figure 6.3: Implementation overview: Unified execution model for CDE

6.4 Tracing Chains for the Precomputation Thread and the Critical Stream

6.4.1 Critical Count Tables

The Critical Count Tables need to track two sets of hard-to-predict branches and long-latency loads as discussed in Section 6.2. Rather than build four separate tables, CDE uses two tables with two sets of counters (Figure 6.4). The Critical-Branch Count Table holds two counters with different decrement periods. The first counter captures hard-to-predict branches for the precomputation thread - it has a lower decrement period and is thus more selective about which branches are marked. The second counter captures hard-to-predict branches for the critical stream and has a high decrement period to ensure good coverage. The Critical Load Count Table is similarly modified to contain two counters. The precomputation thread counter is only incremented if an LLC miss is seen and its latency is greater than the current run-ahead distance of the precomputation thread. This allows the precomputation thread to selectively accelerate loads that can be prefetched. An entry in either table is replaced only if both its counter values are 0.

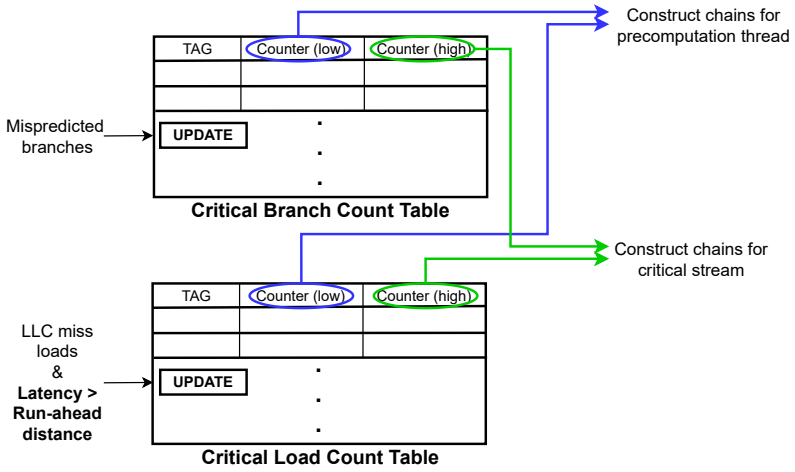

Figure 6.4: Critical Count Tables with two counters

6.4.2 Fill Buffer

The Fill Buffer is modified to contain two sets of bits indicating whether the uop is a hard-to-predict branch, a long-latency load, and whether it is part of the chains, for the precomputation thread and the critical stream separately. The rest of the bits remain the same.

6.4.3 Backward Dataflow Walk

The Backward Dataflow Walk is performed twice when the Fill Buffer is full. In the first instance, chain uops for the precomputation thread are traced using the corresponding bits for hard-to-predict branches and long-latency loads in the Fill Buffer. This marks the chain bit for precomputation thread uops. The second instance of the Walk traces chain uops for the critical stream. However, hard-to-predict branches or long-latency loads already in the precomputation thread are not used as initiation points. This ensures the precomputation thread and critical stream don't accelerate the same chains. Figure 6.5 demonstrates this process with an example.

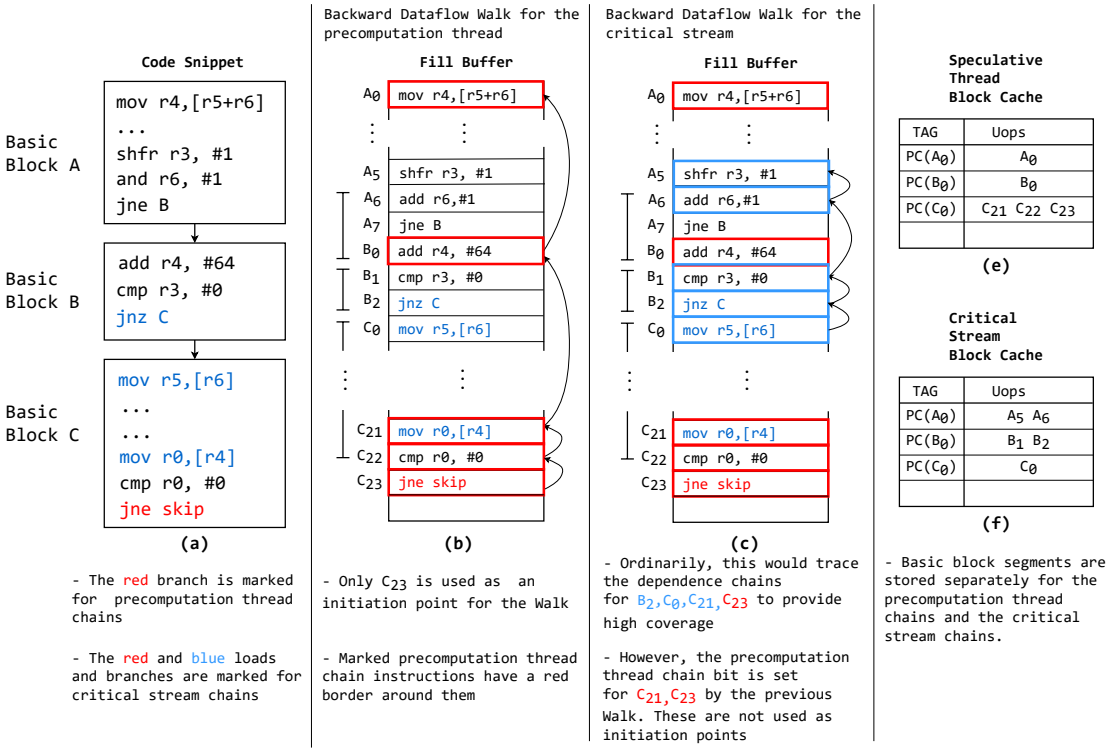

Figure 6.5: Tracing chains in the combined model

6.5 Dynamically Adjusting how Chains are Accelerated

The initial parameters for the Critical Count Tables were obtained by sweeping various configurations and are listed in Figure 6.6.a. The critical stream decrement periods are initialized to 100K.

To account for the variability in backend contention among benchmarks, the decrement period for the precomputation thread counter in the Critical-Branch Count Table is dynamically adjusted (Figure 6.6.b). This mechanism measures the IPC in an epoch (100K retired instructions in my implementation) with the current decrement period and half the decrement period for the branch counter. The decrement period is updated if the difference in IPC between the two configurations is greater than 0.2. If the current period performs significantly better (IPC difference $>$ 0.4), the decrement period is opportunistically increased. If halving the period performs better,

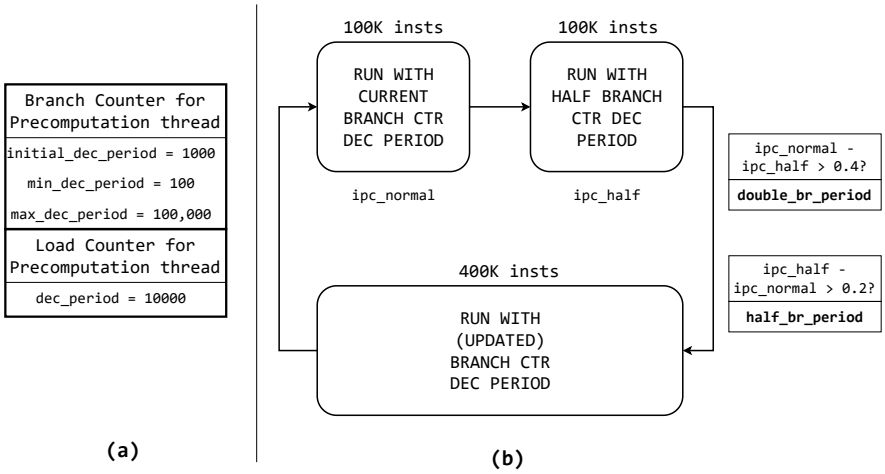

Figure 6.6: Dynamically adjusting the decrement period for precomputation thread branches and loads

the decrement period is decreased. If the decrement period goes below a certain threshold, no chains are traced for precomputation thread branches. Only the branch decrement period is varied since the precomputation thread selectively picks which long-latency load chains to accelerate by looking at the current run-ahead distance.

6.6 Frontend Changes for the Unified Model

6.6.1 Fetch

The Fetch stage for the critical stream is not combined with the main pipeline to avoid re-fetching instructions multiple times. The critical stream Block Cache can be optimized to store just masks instead of uops, reducing its storage at the cost of increased latency for the critical stream (as critical instructions would need to be fetched from the I-Cache and then decoded). This drops performance by $\sim 0.7\%$ but reduces the storage cost of the critical Block Cache from 19KB to 3KB.

6.6.2 Rename

Rename uses three individual RATs to manage dependencies for the three instruction streams. The main RAT contains two poison bits to track incorrect chains simultaneously for the precomputation thread and the critical stream. The bit-masks for both sets of chains provide this information (as was the case for the individual implementations).

6.6.3 Allocation

Allocation for the precomputation thread uses 4 dedicated write ports into the Reservation Stations. The 8 baseline allocation ports are shared by the precomputation thread and the main thread, with priority given to the precomputation thread. Arbitration between the critical and non-critical streams is done at the instruction buffers in this implementation, with priority given to critical stream uops.

6.7 Backend Changes for the Unified Model

Physical Registers and Reservation Stations are allocated on a first-come-first-served basis. 32 entries are reserved for the precomputation thread, critical stream, and non-critical stream (when active) to ensure each of them makes sufficient forward progress. In addition, a limit of 300 Physical Registers is set for the precomputation thread.

The ROB, LQ, and SQ are partitioned between the critical and non-critical streams as before, and the dynamic partitioning algorithm remains unchanged.

6.8 Hardware Overhead

Area and Power: The hardware overhead for this implementation increases slightly compared to the two individual models since it has two Block Caches and two additional RATs. This puts the area overhead at 4.9% and the additional peak

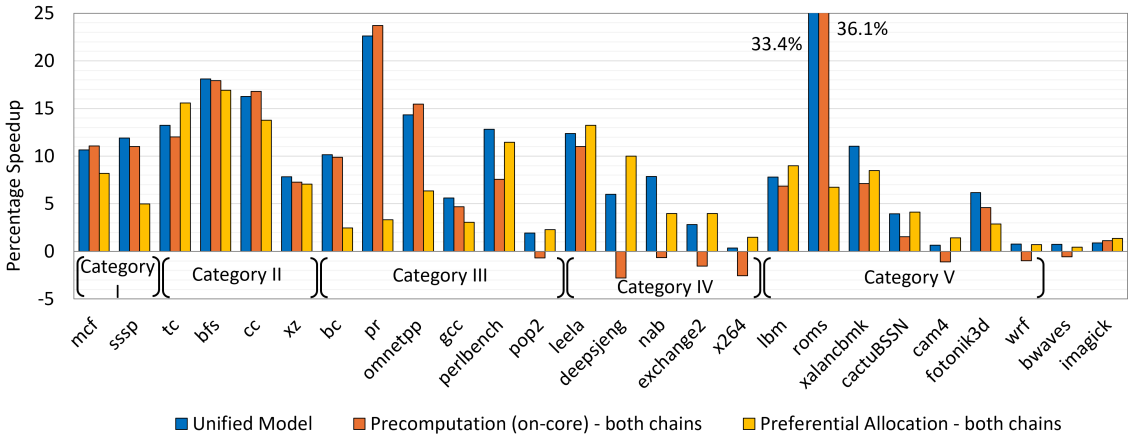

Figure 6.7: Speedup of the unified execution model

power at 10% over the baseline OoO core.

Energy: The unified model sits mid-way between preferential allocation and precomputation. However, since most of the chains are handled by the critical stream and the overall runtime is reduced significantly, it reduces energy consumption by 4.3%.

6.9 Evaluation

6.9.1 Performance

Figure 6.7 shows the percentage speedup of the unified model over the baseline. It also shows the best-performing preferential allocation and on-core precomputation configurations, targeting both types of chains. The period-adjusting mechanism and Critical Count Table parameters were tuned separately for the on-core precomputation thread for best geomean performance. The Critical Count Tables were tuned for preferential allocation to maximize coverage.

In the category I and II benchmarks, the models perform closely, with preferential allocation losing out in some cases due to its limited run-ahead distance. Since most hard-to-predict chains contain long-latency loads in these benchmarks, they do

not see much improvement over just using hard-to-predict branch chains (Figure 6.2).

Category III is where combining the chains shows the most benefit. *bc*, *pr*, and *omnetpp* all perform significantly better with both chains on all execution models. The unified model does much better on *gcc*, *perlbench*, and *pop2* as it can effectively split off some chains to the critical stream in these benchmarks.

Precomputation does poorly on category IV and some category V benchmarks due to contention on the Floating Point, Vector, Branch Functional Units, and D-cache ports. Conversely, preferential allocation performs well on these benchmarks, and the unified model improves on it by accelerating a few chains via precomputation.

pr, bc and *roms* benefit significantly from the load prefetching effect of the precomputation thread due to their large run-ahead distance.

Overall comments: The unified model provides better performance (9% geomean) and has no negative outliers. The combined precomputation thread with optimizations provides a 7% geomean speedup. Without the late precomputation terminations and dynamic period changes, the performance of this configuration decreases by 2%. Preferential allocation only improves performance by 6.1%, but has the lowest overhead. On average, no additional instructions enter the backend in preferential allocation. For precomputation, this number is 20%. In the unified model, on average 12% more instructions enter the processor backend.

The precomputation thread narrowly outperforms the unified model in several benchmarks. This is due to the variable decrement period. The mechanism sometimes greedily removes branches from the precomputation thread (especially in category I and II benchmarks), and performs worse in the long term. Preferential allocation also performs better than the unified model in some cases where minimizing backend contention is more important.

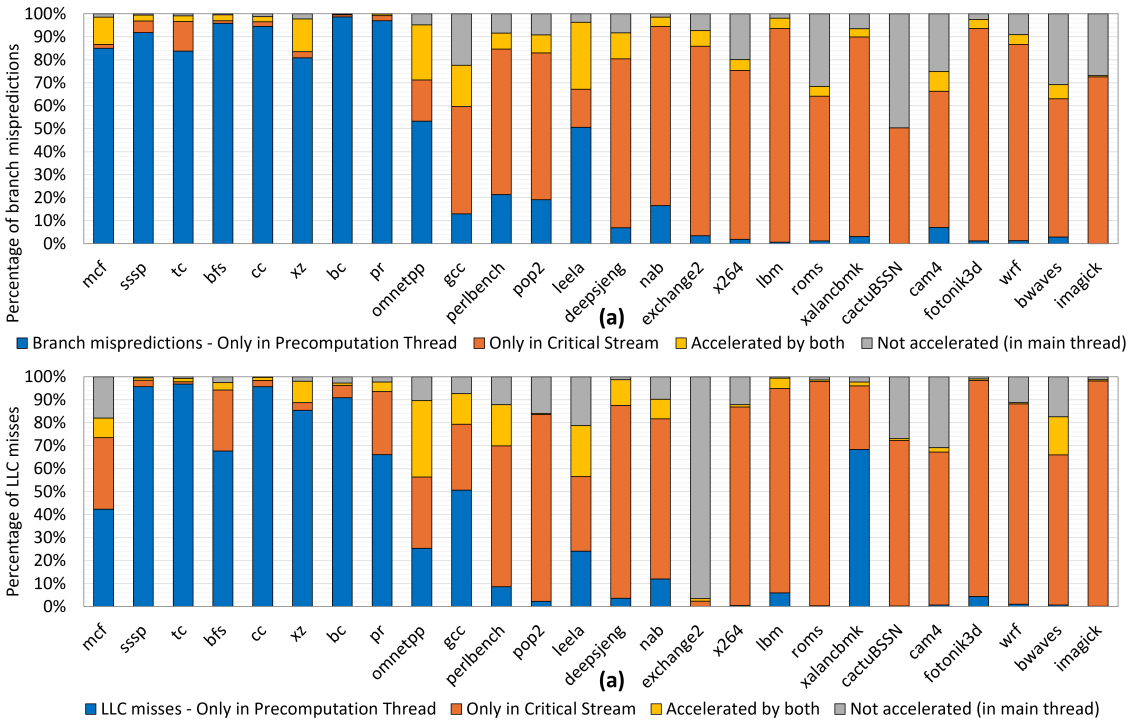

Figure 6.8: Branch misprediction and LLC miss coverage

6.9.2 Misprediction Coverage

In the unified model, the precomputation thread covers most branch mispredictions and LLC misses in category I and II benchmarks as seen in Figure 6.8. The critical stream covers the bulk of the misses and mispredictions in the rest of the benchmarks, as the critical stream performs comparably (or better) on these branches and loads. A small portion of branch mispredictions and LLC misses are covered by both the precomputation thread and the critical stream. The masks in the Block Cache are responsible for this. When a chain is removed from the precomputation thread, it remains in the Block Cache masks until the masks are eventually reset (every 500k instructions).

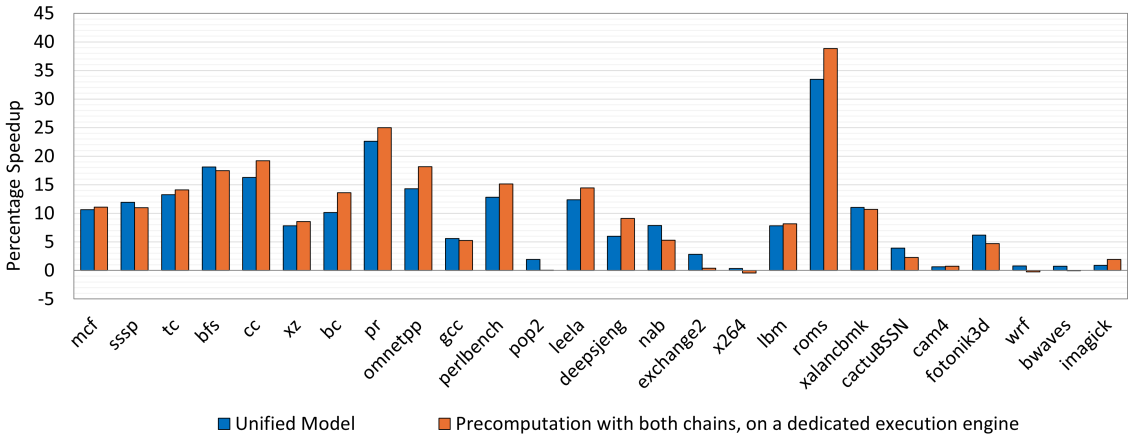

Figure 6.9: Comparison against a Slipstream-Like approach

6.9.3 Comparison against a Slipstream-Like Approach

Using a separate core increases communication latency and drastically reduces the branch misprediction penalty saved. Instead, I used a separate OoO execution engine containing the same number of Reservation Stations, Physical Registers, and Functional Units as the main core (with shared D-cache ports and MSHRs) to model a Slipstream-Like approach. This engine only runs the precomputation thread with both hard-to-predict and long-latency load chains, eliminating any performance loss due to backend contention (except for D-cache ports). The Critical Count Table and watchdog parameters were optimized for this configuration.Figure 6.9 shows its speedup. The geomean performance improvement is 9.5% over the baseline OoO core.

The dedicated execution performs marginally better on most benchmarks. The unified model does better on a few benchmarks by dividing chains between the precomputation thread and critical stream to provide better fetch bandwidth for the precomputation thread. For instance, *bfs*, *sssp*, and *pop2* perform worse with a combined precomputation thread. **The dedicated engine consumes 1.5x more power, 1.35x more area, and 1.29x more energy compared to the baseline OoO core, making it significantly less efficient.**

6.9.4 Parameter Tuning

The unified model relies on the guidelines laid out in Section 6.2.2 to separate out the load and branch chains for best performance, and to pick the best performing configuration between preferential allocation and precomputation when the chains cannot be separated. The key parameters for tuning this are the decrement period for the Critical Count Tables, the threshold at which loads are assigned to the precomputation thread, and the measurement period of the contention-monitoring feedback mechanism.

The Critical Count Table parameters follow the same trends as the individual execution models and are the most important to tune, with a variance of 2%-3% in speedup across all benchmarks. The threshold at which loads are assigned to the precomputation thread mainly impacts category IV benchmarks (with roms seeing a loss of 4%-5% speedup if too few loads are added to the precomputation thread). The contention-monitoring feedback mechanism parameters have a much lower overall impact, less 1% difference in the geomean speedup. However, tuning them removes negative outliers (*pop2, bwaves, cam4, x264*).

Chapter 7: Conclusion and Future Work

7.1 Conclusion

Single-thread performance has come a long way in the past 30 years. Branches and loads, which were deemed significant bottlenecks when OoO execution was first introduced, have improved with better branch prediction and data prefetching algorithms. However, as modern designs continue to push for wider and deeper cores, the impact of branch mispredictions and cache misses on performance continues to grow. Even with commercially implemented algorithms, eliminating these mispredictions and misses can provide over a 2x IPC improvement. The fundamental challenge lies in coverage— existing approaches only address a small fraction of branch mispredictions and cache misses, particularly those caused by hard-to-predict branches and long-latency loads.

While eliminating the full penalty of branch mispredictions and cache misses is challenging, issuing early misprediction flushes and launching multiple long-latency loads in parallel can significantly reduce their performance impact. This approach enables the mitigation of a broader class of mispredictions and cache misses that prior techniques fail to address, which is the main focus of this dissertation.

At the heart of Criticality Driven Execution (CDE) is its thread construction mechanism, which dynamically traces highly accurate, long, and lightweight dependence chains for both branches and loads. These chains are annotated with timestamps that establish their ordering relative to the rest of the instruction stream to simplify communication. CDE leverages these chains in two ways: building a speculative precomputation thread that accelerates the resolution of hard-to-predict branch chains, and using a preferential allocation scheme that improves Memory Level Parallelism for long-latency loads. These mechanisms operate in tandem to utilize existing on-chip resources more effectively without adding extra execution hardware, achiev-

ing an overall 9.0% IPC improvement over an aggressive OoO core for the SPEC CPU2017 and GAP benchmark suites.

This work demonstrates that redistributing existing OoO core resources to prioritize critical instructions is an effective and efficient strategy for improving performance. By dynamically adapting fetch and allocation priorities based on instruction criticality, CDE can scale frontend bandwidth and instruction window size for instructions important to performance without the exponential cost of building a wider pipeline or a deeper backend.

7.2 Future Work

The execution models proposed in this dissertation require several dynamic feedback mechanisms. They interact with each other in complex ways and need to be driven by targeted feedback from the OoO core beyond just stall cycles and IPC. A holistic mechanism that combines all these aspects and partitions allocation bandwidth and backend resources is key to extracting all the performance benefits that CDE could not uncover. This can drive a more targeted design for an OoO core, where the parameters are decided by how to best accelerate critical chains rather than targeting blanket fetch bandwidth and parallelism improvements for all instructions.

Besides direct improvements, the findings of this dissertation allow for more exploration in other aspects of microarchitecture.

Using the Chain construction mechanism: CDE's chains are highly accurate and long while containing as few instructions as possible to maintain this high accuracy [13, 14]. This opens up opportunities for isolating important program segments that can be used to improve other aspects of OoO core performance, such as Instruction Level Parallelism (ILP).

Adopting Preferential Allocation: Preferential allocation [14] provides a means for fetching, renaming, and allocating instructions out-of-order. Not all these

operations need to be performed out-of-order to prioritize instructions. For instance, a simpler implementation may only rename and allocate instructions to Reservation Stations and Physical Registers out-of-order while buffering the remaining instructions in the ROB or cheaper in-order queues. This enables better scheduling without modifying the scheduling logic in the Reservation Stations, which is often latency-sensitive.

Using control flow to accelerate loads: The execution models in this dissertation show that accounting for hard-to-predict branch chains is necessary for accelerating loads in many workloads [13]. This suggests that incorporating accurate control flow information from the main branch predictor is essential for accurately identifying future loads for tasks such as prefetching. A simple branch predictor or static prediction is insufficient for applications with complex control flows, which high-performance OoO cores are often geared towards.

References

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

[2] "The standard performance evaluation corporation (spec)," 1997. [Online]. Available: https://www.spec.org/

[3] M. Agarwal, N. Navale, K. Malik, and M. I. Frank, "Fetch-Criticality Reduction through Control Independence," in *2008 International Symposium on Computer Architecture*, 2008.

[4] H. Akkary, R. Rajwar, and S. T. Srinivasan, "Checkpoint processing and recovery: Towards scalable large instruction window processors," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 36. USA: IEEE Computer Society, 2003, p. 423.

[5] R. Balasubramonian, S. Dwarkadas, and D. Albonesi, "Dynamically allocating processor resources between nearby and distant ilp," in *28th Annual International Symposium on Computer Architecture (ISCA)*, 2001.

[6] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," 2017. [Online]. Available: https://arxiv.org/abs/1508.03619

[7] B. Black, B. Rychlik, and J. Shen, "The block-based trace cache," in *Proceedings of the 26th International Symposium on Computer Architecture (Cat. No.99CB36367)*, 1999, pp. 196–207.

[8] T. E. Carlson, W. Heirman, O. Allam, S. Kaxiras, and L. Eeckhout, "The load slice core microarchitecture," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015, pp. 272–284.

[9] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt, "Simultaneous subordinate microthreading (ssmt)," in *Proceedings of the 26th International Symposium on Computer Architecture (Cat. No.99CB36367)*, 1999, pp. 186–195.

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

[11] A. Chauhan, J. Gaur, Z. Sperber, F. Sala, L. Rappoport, A. Yoaz, and S. Subramoney, "Auto-predication of critical branches," in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ser. ISCA '20. IEEE Press, 2020, p. 92–104. [Online]. Available: https://doi.org/10.1109/ISCA45697.2020.00019

[12] A. Deshmukh, L. C. Cai, and Y. N. Patt, "Alternate path fetch," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024, pp. 1217–1229.

[13] A. Deshmukh, L. Cai, and Y. N. Patt, "Timely, efficient, and accurate branch precomputation," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2024, pp. 480–492.

[14] A. Deshmukh and Y. N. Patt, "Criticality driven fetch," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 380–391. [Online]. Available: https://doi.org/10.1145/3466752.3480115

[15] J. Doweck, "Inside intel core microarchitecture and smart memory access," 2006. [Online]. Available: https://www.intel.com/pressroom/kits/core2duo/pdf/ICM_whitepaper.pdf

[16] Q. Duong, A. Jain, and C. Lin, "A new formulation of neural data prefetching," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024, pp. 1173–1187.

[17] A. Farcy, O. Temam, R. Espasa, and T. Juan, "Dataflow analysis of branch mispredictions and its application to early resolution of branch outcomes," in *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*, 1998, pp. 59–68.

[18] D. Friendly, S. J. Patel, and Y. Patt, "Alternative fetch and issue policies for the trace cache fetch mechanism," in *Proceedings of 30th Annual International Symposium on Microarchitecture*, 1997, pp. 24–33.

[19] A. Garg and M. C. Huang, "A performance-correctness explicitly-decoupled architecture," in *41st International Symposium on Microarchitecture (MICRO)*, 2008.

[20] B. Grayson, J. Rupley, G. Z. Zuraski, E. Quinnell, D. A. Jiménez, T. Nakra, P. Kitchin, R. Hensley, E. Brekelbaum, V. Sinha, and A. Ghiya, "Evolution of the samsung exynos cpu microarchitecture," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 40–51.

[21] S. Gupta, N. Soundararajan, R. Natarajan, and S. Subramoney, "Opportunistic early pipeline re-steering for data-dependent branches," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '20, 2020, p. 305–316. [Online]. Available: https://doi.org/10.1145/3410463.3414628

[22] M. Hashemi and Y. N. Patt, "Filtered runahead execution with a runahead buffer," in *48th International Symposium on Microarchitecture (MICRO)*, 2015.

[23] M. Hashemi, O. Mutlu, and Y. N. Patt, "Continuous Runahead: Transparent Hardware Acceleration for Memory Intensive Workloads," in *49th International Symposium on Microarchitecture (MICRO)*, 2016.

[24] Huiyang Zhou, "Dual-core execution: building a highly scalable single-thread instruction window," in *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, 2005.

[25] S. Iacobovici, L. Spracklen, S. Kadambi, Y. Chou, and S. G. Abraham, "Effective stream-based and execution-based data prefetching," in *Proceedings of the 18th Annual International Conference on Supercomputing*, ser. ICS '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 1–11. [Online]. Available: https://doi.org/10.1145/1006209.1006211

[26] D. Jiménez, "Multiperspective perceptron predictor," in *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, 2016.

[27] Jiwei Lu, A. Das, Wei-Chung Hsu, Khoa Nguyen, and S. G. Abraham, "Dynamic helper threaded prefetching on the Sun UltraSPARC/spl reg/ CMP processor," in *38th International Symposium on Microarchitecture (MICRO'05)*, 2005.

[28] T. A. Khan, M. Ugur, K. Nathella, D. Sunwoo, H. Litz, D. A. Jiménez, and B. Kasikci, "Whisper: Profile-guided branch misprediction elimination for data center applications," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 19–34.

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

[30] S. Kondguli and M. Huang, "Bootstrapping: Using SMT Hardware to Improve Single-Thread Performance," in *ASPLOS '19*, 2019.

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

[32] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42, 2009, p. 469–480. [Online]. Available: https://doi.org/10.1145/1669112.1669172

[33] H. Litz, G. Ayers, and P. Ranganathan, "Crisp: critical slice prefetching," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 300–313. [Online]. Available: https://doi.org/10.1145/3503222.3507745

[34] C. Madriles, P. López, J. M. Codina, E. Gibert, F. Latorre, A. Martinez, R. Martinez, and A. Gonzalez, "Boosting Single-Thread Performance in Multi-Core Systems through Fine-Grain Multi-Threading," in *36th Annual International Symposium on Computer Architecture (ISCA)*, 2009.

[35] K. Malik, M. Agarwal, S. S. Stone, K. M. Woley, and M. I. Frank, "Branch-mispredict level parallelism (blp) for control independence," in *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, 2008, pp. 62–73.

[36] M. Moudgill, K. Pingali, and S. Vassiliadis, "Register renaming and dynamic speculation: an alternative approach," in *Proceedings of the 26th Annual International Symposium on Microarchitecture*, ser. MICRO 26. Washington, DC, USA: IEEE Computer Society Press, 1993, p. 202–213.

[37] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: an alternative to very large instruction windows for out-of-order processors," in *9th International Symposium on High-Performance Computer Architecture, (HPCA)*, 2003.

[38] A. Naithani, J. Feliu, A. Adileh, and L. Eeckhout, "Precise Runahead Execution," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[39] A. Naithani, S. Ainsworth, T. M. Jones, and L. Eeckhout, "Vector runahead," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 195–208.

[40] A. Naithani, J. Roelandts, S. Ainsworth, T. M. Jones, and L. Eeckhout, "Decoupled vector runahead," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 17–31. [Online]. Available: https://doi.org/10.1145/3613424.3614255

[41] A. Navarro-Torres, B. Panda, J. Alastruey-Benedé, P. Ibáñez, V. Viñals-Yúfera, and A. Ros, "Berti: an accurate local-delta data prefetcher," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 975–991.

[42] S. Pakalapati and B. Panda, "Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 118–131.

[43] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-effective superscalar processors," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ser. ISCA '97. New York, NY, USA: Association for Computing Machinery, 1997, p. 206–218. [Online]. Available: https://doi.org/10.1145/264107.264201

[44] B. Panda, "Clip: Load criticality based data prefetching for bandwidth-constrained many-core systems," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 714–727. [Online]. Available: https://doi.org/10.1145/3613424.3614245

[45] S. Patel, M. Evers, and Y. Patt, "Improving trace cache effectiveness with branch promotion and trace packing," in *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No.98CB36235)*, 1998, pp. 262–271.

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

[47] G. Reinman, B. Calder, and T. Austin, "Fetch directed instruction prefetching," in *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, 1999, pp. 16–27.

[48] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace cache: a low latency approach to high bandwidth instruction fetching," in *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 29. USA: IEEE Computer Society, 1996, p. 24–35.

[49] E. Safi, A. Moshovos, and A. Veneris, "Two-stage, pipelined register renaming," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 10, pp. 1926–1931, 2011.

[50] A. Sembrant, T. Carlson, E. Hagersten, D. Black-Shaffer, A. Perais, A. Seznec, and P. Michaud, "Long term parking (LTP): Criticality-aware resource allocation in OOO processors," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015.

[51] A. Seznec, "A new case for the tage branch predictor," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44, 2011, p. 117–127. [Online]. Available: https://doi.org/10.1145/2155620.2155635

[52] ——, "Tage-sc-l branch predictors again," in *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, 2016.

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

[54] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," ser. ASPLOS X, 2002, p. 45–57. [Online]. Available: https://doi.org/10.1145/605397.605403

[55] Z. Shi, A. Jain, K. Swersky, M. Hashemi, P. Ranganathan, and C. Lin, "A hierarchical neural model of data prefetching," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '21, 2021, p. 861–873. [Online]. Available: https://doi.org/10.1145/3445814.3446752

[56] F. M. Sleiman and T. F. Wenisch, "Efficiently scaling out-of-order cores for simultaneous multithreading," in *43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[57] S. Somogyi, T. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial memory streaming," in *33rd International Symposium on Computer Architecture (ISCA'06)*, 2006, pp. 252–263.

[58] S. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton, "Continual flow pipelines: Achieving resource-efficient latency tolerance," *IEEE Micro*, 2004.

[59] V. Srinivasan, R. B. R. Chowdhury, and E. Rotenberg, "Slipstream processors revisited: Exploiting branch sets," in *47th International Symposium on Computer Architecture (ISCA)*, 2020.

[60] K. Sundaramoorthy, Z. Purser, and E. Rotenberg, "Slipstream processors: Improving both performance and fault tolerance," *SIGPLAN Not.*, vol. 35, no. 11, Nov. 2000.

[61] ——, "Slipstream processors: Improving both performance and fault tolerance," *SIGPLAN Not.*, vol. 35, no. 11, p. 257–268, nov 2000. [Online]. Available: https://doi.org/10.1145/356989.357013

[62] K. Tran, T. E. Carlson, K. Koukos, M. Själander, V. Spiliopoulos, S. Kaxiras, and A. Jimborean, "Clairvoyance: Look-ahead compile-time scheduling," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2017.

[63] P. H. Wang, J. D. Collins, H. Wang, D. Kim, B. Greene, K.-M. Chan, A. B. Yunus, T. Sych, S. F. Moore, and J. P. Shen, "Helper threads via virtual multithreading on an experimental itanium® 2 processor-based platform," ser. ASPLOS XI. New York, NY, USA: Association for Computing Machinery, 2004, p. 144–155. [Online]. Available: https://doi.org/10.1145/1024393.1024411

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

[65] C. Zilles and G. Sohi, "Execution-based prediction using speculative slices," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ser. ISCA '01. New York, NY, USA: Association for Computing Machinery, 2001, p. 2–13. [Online]. Available: https://doi.org/10.1145/379240.379246

[66] C. B. Zilles and G. S. Sohi, "Understanding the backward slices of performance degrading instructions," ser. ISCA '00. New York, NY, USA: Association for Computing Machinery, 2000, p. 172–181. [Online]. Available: https://doi.org/10.1145/339647.339676