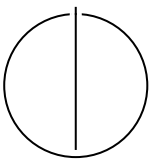
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Evaluation of Multiple Branch Prediction's Potential in the Context of Wide-Pipeline Architectures

Steve Bambou Biangang



SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Evaluation of Multiple Branch Prediction's Potential in the Context of Wide-Pipeline Architectures

Bewertung des Potenzials der Mehrfachzweigvorhersage im Kontext von Wide-Pipeline-Architekturen

Author:	Steve Bambou Biangang
Examiner:	Prof. Dr. Pramod Bhatotia
Supervisor:	Dr. David Schall
Submission Date:	28.08.2025

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 28.08.2025 Steve Bambou Biangang

Acknowledgments

I want to start by thanking my advisor, David, for giving me the opportunity to work on this exciting project and for always being available to answer my questions. I want to thank my family for their support this semester, who constantly check on me to ensure I'm okay and motivate me. I would also like to thank my friends for the fun moments we spent together this semester. It was really nice to disconnect a bit from the thesis with them. Special thanks to my housemate, Simon, for giving me helpful tips for scientific writing.

Abstract

As pipelines grow wider and instruction windows grow larger over the years, scalable instruction fetching mechanisms become crucial to overcome the instruction supply bottleneck. While prior research on instruction-level parallelism (ILP) showed significant performance improvements by scaling instruction window and pipeline width, these studies lacked modeling of critical bottlenecks and the frontend capabilities necessary to ensure that the core can effectively fetch sufficient instructions to fill the instruction window.

This work addresses the need for realistic performance evaluation of wide-pipeline processors by investigating three interconnected research questions. First, how does multiple branch prediction combined with fetch-directed instruction prefetching (FDP) impact instruction fetch rates as pipeline width scales? Second, can wider instruction windows deliver the performance improvements suggested by prior ILP studies when comprehensive system modeling is employed? Third, which critical backend components are most susceptible to creating bottlenecks under increased speculative execution?

To answer these questions, we implement multiple branch prediction in gem5's O3 core and systematically evaluate performance using SPEC integer benchmarks and server workloads. We then optimize the backend configuration of the core to improve the IPC and approach the exploitable ILP limit.

The effectiveness of our frontend design is demonstrated by the instruction fetch rate we achieved on the scaled architecture: 15.2 on average for six predictions per cycle. Compared to the 4.41 fetch rate for one prediction, this represents a 3.4X speedup. Moreover, we identified the recovery penalty as the most significant bottleneck of the increased speculative execution enabled by larger instruction windows. Regarding ILP, we achieved an average of 7.15 IPC with six predictions per cycle, compared to 3.89 for one prediction. This result demonstrates the potential of multiple branch prediction for wide-pipeline architectures and provides an estimate of the ILP limit when considering real hardware bottlenecks. The full implementation in gem5 is available here: `https://github.com/dhschall/gem5-fdp/tree/pf-rework-multi-pred`

Contents

1 Introduction

Programs inherently exhibit instruction-level parallelism (ILP), meaning that independent instructions within a program can be identified and executed simultaneously without affecting correctness. To exploit this available parallelism and significantly speed up execution times, modern processor architectures rely on superscalar out-of-order execution techniques that dynamically reorder and dispatch multiple instructions per cycle. Current microarchitectural trends demonstrate growing pipeline width and increasingly large instruction windows, enabling processors to extract and utilize greater amounts of ILP from application code [8, 20].

However, exploiting this ILP effectively depends on the processor's ability to fetch instructions at sufficient rates. As modern processors become increasingly capable of executing multiple instructions per cycle, the bottleneck shifts from execution capacity to instruction supply. While memory latency has been reducing over the years, accessing instructions from memory remains significantly slower than processor execution speeds [9]. When growing the pipeline width and instruction window gives the processor's backend more room to exploit ILP by handling more instructions simultaneously, this increased execution capacity is not sufficient if the frontend cannot fetch enough instructions fast enough to keep the execution units fed. This mismatch between execution capability and instruction supply creates the need for scalable instruction fetching mechanisms that can deliver instructions at rates matching the processor's execution throughput.

To address these instruction supply challenges, several techniques have been developed to improve fetch performance. Fetch-directed instruction prefetching (FDP) [25, 26] is a state-of-the-art mechanism designed to hide instruction cache miss latency by proactively fetching instructions before they are needed. This technique relies on the branch prediction unit running ahead of the fetch stage to predict future instruction addresses and initiate early memory requests. It is well established that this prefetching approach can substantially improve instructions per cycle (IPC) by reducing stall cycles caused by cache misses, though this comes at the cost of higher cache bandwidth utilization due to speculative memory traffic.

Multiple branch prediction [32, 36, 40] is a technique designed to improve the instruction fetch rate by predicting multiple branches and fetching from multiple cache lines simultaneously. For programs characterized by small basic blocks and frequent

branches, multiple branch prediction can significantly increase the core's instruction throughput. Both FDP and multiple branch prediction enable faster instruction supply to the execution units, but these techniques have only been evaluated on narrow-width processor architectures with limited execution resources.

In parallel with these fetch mechanism developments, studies on instruction-level parallelism have been conducted to estimate the influence of instruction window size and other components (branch prediction, pipeline width) on processor performance, specifically on IPC. These studies consistently demonstrate that larger instruction windows can lead to improved performance by providing more opportunities to find independent instructions [35, 10]. While these ILP studies provide valuable estimations of theoretical parallelism limits, they omit the impact of other critical processor components. Specifically, they lack realistic modeling of instruction supply. The core's frontend, responsible for filling the growing instruction windows, is not considered. Moreover, these studies typically omit critical bottlenecks such as cache misses and branch misprediction recovery penalties that introduce additional latency in real systems. To achieve a more realistic assessment of wide-pipeline performance potential, all these surrounding components and their interactions must be taken into account.

This reveals a critical research gap: while scalable frontend designs are needed to support growing pipeline width, the effect of techniques like multiple branch prediction on wide-pipeline architectures remains unstudied. Furthermore, most previous ILP studies have not accounted for frontend configuration impacts and other critical components, resulting in performance assessments that overlook the instruction supply and other bottlenecks. While this is a feature of these studies, an assessment with a comprehensive model is also necessary to understand the effect of these bottlenecks.

This work addresses the need for realistic performance evaluation of wide-pipeline processors by investigating three interconnected research questions. First, how does multiple branch prediction combined with FDP impact instruction fetch rates as pipeline width scales? Second, can wider instruction windows deliver the performance improvements suggested by prior ILP studies when comprehensive system modeling is employed? Third, which critical backend components are most susceptible to creating bottlenecks under increased speculative execution?

To address these questions, we employ a simulation-based approach using gem5, one of the most widely contributed and utilized microarchitecture simulators. Gem5 is particularly suited for this research as it models the complete CPU pipeline behavior and provides execution-based simulation capabilities. Our methodology consists of three key components: First, we implement multiple branch prediction in the O3 core, which is currently missing, extending the already existing FDP implementation [27]. Second, we evaluate the effect of multiple branch prediction with FDP on the instruction

fetch rate for wide-pipeline architectures. Third, we analyze the simulation results to identify and highlight critical backend components that become bottlenecks under increased instruction throughput. Fourth, we optimize backend configurations of the gem5's O3 core to maximize multiple branch prediction's IPC improvement and establish realistic ILP estimates for wide-pipeline architectures.

We conduct a comprehensive performance evaluation across two dimensions. Using a representative mix of SPEC integer benchmarks [34] and server workloads [15, 28, 7], we first assess how multiple branch prediction performs on current and wide-pipeline architectures, with particular focus on instruction throughput gains and Top-Down methodology frontend bound metrics. Subsequently, we explore different backend configuration scenarios with the same benchmark suite to determine maximum IPC potential.

This thesis provides insights into the potential of multiple branch prediction with FDP as a scalable frontend design capable of maximizing instruction throughput while minimizing latency. Additionally, our ILP-maximizing approach for the frontend can guide future work toward possible optimizations of the O3 core backend.

This thesis makes four key contributions. We implement multiple branch prediction in gem5. Through systematic evaluation, we demonstrate the performance potential of multiple branch prediction for wide-pipeline processor architectures. We provide realistic assessments of instruction-level parallelism limits by evaluating multiple branch prediction's impact on backend components using a detailed model that accounts for recovery penalties, cache misses, pipeline stalls, and other critical bottlenecks. Additionally, we identify the current limitations of gem5's O3 core that impact the simulation accuracy in comparison to real hardware.

2 Background

This Chapter introduces the relevant background necessary to better understand the core of this thesis. We first introduce the important concepts of out-of-order execution. Then, we go over the concept of branch prediction and introduce TAGE [33] as a state-of-the-art branch predictor. Subsequently, we present PHAST [18, 19] as a state-of-the-art memory dependency predictor. Then, we introduce multiple branch prediction [32, 36, 40], fetch-directed instruction prefetching (FDP) [25, 26], gem5, the microarchitecture simulator we use for this study, and the O3 core's pipeline. Finally, we introduce the Top-Down methodology [39], a state-of-the-art tool for bottleneck analysis.

2.1 Out-of-Order Processor

ILP is a characteristic of a program expressing how many instructions can be executed simultaneously on average without modifying the program's behavior. It expresses the potential execution speed of the program. In-order processors fail to exploit the maximum ILP due to their sequential execution constraint. In comparison, out-of-order processors, which allow multiple instruction issue per cycle and out-of-order execution, enable better leveraging of ILP. Out-of-order execution relies on multiple techniques and structures:

- Superscalar execution: This refers to a processor's ability to execute multiple instructions simultaneously within a single clock cycle. Unlike scalar processors that handle one instruction at a time, superscalar processors have multiple execution units (ALUs, floating-point units, load/store units, etc.) that can operate in parallel. For example, a 4-way superscalar processor can theoretically execute up to 4 instructions per cycle, significantly increasing throughput compared to single-issue processors.

- Register renaming: This technique solves the problem of false dependencies (also called WAR - Write After Read, and WAW - Write After Write dependencies). When multiple instructions use the same architectural register name but don't have a true data dependency, the processor can map them to different physical registers. For instance, if instruction A writes to register R1 and later instruction

B also writes to R1 (but doesn't depend on A's result), register renaming allows both to execute in parallel, using separate physical registers behind the scene.

- The reorder buffer (ROB): The ROB is a crucial structure that maintains the original program order for instruction completion, even when instructions execute out of order. It acts as a circular queue that stores instruction results until they can be committed in program order. The ROB also enables speculative execution by providing a mechanism to discard incorrectly executed instructions.
- Memory dependency predictor (MDP): Since memory operations (loads and stores) can have complex dependencies that aren't known until runtime, processors use prediction mechanisms to speculate about these relationships. Store-to-load forwarding, memory disambiguation, and load/store queues help predict when a load instruction depends on a previous store. Accurate prediction allows loads to execute speculatively before prior stores are resolved, improving performance.
- Instruction scheduling: This is the heart of out-of-order execution. The scheduler or instruction queue (IQ) tracks instruction dependencies and resource availability to determine which instructions are ready to execute. It maintains a pool of instructions and continuously monitors for instructions whose operands are available and whose required execution units are free. The scheduler then issues these ready instructions to appropriate execution units, maximizing processor utilization while respecting true data dependencies.

Research [14, 17, 23, 13, 3, 41] and industry [37] have been engineering out-of-order processors for decades, and they are nowadays the most widely used architecture in high-performance general-purpose computing.

2.2 TAGE-SC-L

Branches, which represent control instructions can modify the execution path of a program at runtime, creating uncertainty about which instructions the processor needs to fetch next. Waiting until the branch is resolved (executed) is not viable for today's wide and deep pipelines, as this would significantly restrict the instruction throughput. *Branch prediction* is a technique used to predict the target of a branch, allowing the processor to fetch the instructions at the predicted target and keep the pipeline active. This is known as speculative execution. The branch predictor's accuracy must be sufficiently high to make speculative execution actually beneficial, because a misprediction costs several cycles to recover. The branch target buffer (BTB) is a cache storing

information about branches that are useful for the branch predictor. It also uses a return address stack (RAS) to store return addresses that might be useful for prediction and there also exists specialized predictors, for instance loop predictors for loop and indirect predictors for indirect branches. All together this builds the branch prediction unit (BPU), responsible for branch prediction.

TAGE (TAgged GEometric history length), invented by A. Seznec et al. [33], is a state-of-the-art branch predictor. TAGE uses multiple prediction tables with different history lengths arranged in a geometric progression. Each table captures branch patterns at different time scales, from short-term to long-term behavior. The key innovation is its tagged entries that reduce conflicts between different branches, and its intelligent selection mechanism that chooses predictions from the table with the longest relevant history while maintaining simpler backup predictions. TAGE dynamically adapts by monitoring prediction accuracy and can switch between different predictors based on recent performance, allowing it to automatically use the simplest effective predictor for each branch pattern. This design makes TAGE both highly accurate and storage-efficient.

TAGE-SC-L, by A. Seznec [31], is an enhanced version of TAGE, coupled with statistical correctors and loop predictors to further improve accuracy. On the Championship Branch Prediction (CBP-5) training traces, the 64 KB TAGE-SC-L predictor achieves 3.9 mispredictions per kilo instructions (MPKI). A version of TAGE for indirect branch prediction named ITTAGE, by A. Seznec [30], also exist to predict indirect branches more accurately.

2.3 PHAST

As mentioned in Section 2.1, the memory dependency predictor is one of the key structure in out-of-order execution as it predicts runtime dependencies of load instructions on store instructions.

PHAST (PatH-Aware STore-distance), invented by S. Kim et al. is a state-of-the-art memory dependence predictor that achieves high accuracy by making two key observations: loads typically depend on only one store (the youngest conflicting one) rather than sets of stores, and the optimal context for prediction is precisely the execution path from that store to the load—no more, no less. Unlike existing predictors that use predetermined or brute-force history lengths, PHAST dynamically determines the minimum necessary history length for each dependence by tracking N+1 divergent branches (where N is the number of divergent branches between the store and load), then trains and predicts using only this essential context information. [18, 19]

This targeted approach achieves a 14.5 KB implementation that performs within

1.50% of ideal prediction while delivering substantial speedups of 1.29-5.05% over state-of-the-art predictors and reducing mispredictions by 62-70%, demonstrating that understanding the fundamental nature of memory dependencies leads to more effective prediction than simply adapting branch prediction techniques. [18]

2.4 Decoupled frontend

A decoupled frontend, invented by Reinman et al. [25, 26], is an architecture where the branch predictor operates independently from the processor's fetch stage, rather than being tightly coupled to the instruction cache. In this design, the branch predictor runs in a separate stage at the beginning of the pipeline, allowing it to execute far ahead of the Fetch unit and generate fetch targets (FTs) that are queued in the fetch target queue (FTQ). These FTs are subsequently consumed by the Fetch stage as it processes instructions.

The primary advantage of this architecture is fetch-directed instruction prefetching (FDP), by Reinman et al. [25, 26], which leverages the predictive capabilities of the decoupled frontend to significantly improve instruction supply. FDP is a technique that hides instruction cache miss latency by issuing prefetches to the instruction cache directly from the addresses contained in the fetch targets. Since the branch predictor can run well ahead of the actual fetch operations, it can identify future instruction addresses and initiate cache line prefetches before they are actually needed, effectively masking memory access delays.

Beyond FDP, the decoupled frontend architecture offers several additional benefits. Most notably, it enables speculative run-ahead operation where the branch predictor can execute many cycles ahead of the fetch stage, building predictive chains of future branch outcomes and instruction paths before they are needed. This speculative advancement allows the predictor to operate more aggressively with sophisticated algorithms and larger prediction structures, since it is not constrained by the tight timing requirements of instruction cache access. [26]

This architectural approach has become the standard in modern high-performance CPU designs, adopted in research [16] as well as by major processor vendors, including Arm [24] and IBM [1]. The widespread adoption reflects its effectiveness in maximizing instruction throughput and overall processor performance in modern CPU architectures.

2.5 gem5

gem5[1] is an open-source computer architecture simulator, popular in both research and industry [2, 22]. It provides support for multiple instruction set architectures (ISAs), making it a versatile platform for computer systems research. The simulator is execution-driven, meaning it models the actual execution of instructions through the processor pipeline, which typically provides more accurate results compared to trace-driven simulators that replay pre-recorded instruction sequences.

gem5 offers several CPU models with varying levels of complexity and accuracy, ranging from simple atomic models for fast functional simulation to detailed timing models that capture microarchitectural behavior. The simulator also includes comprehensive memory system modeling, supporting multiple cache levels, coherence protocols, and interconnection networks. This flexibility allows researchers to study system performance across different configurations and workloads.

In the course of this work we will focus on the O3 Core. The O3 Core implements out-of-order and speculative execution, originally based on the Alpha 21264 processor. It models key microarchitectural features including branch prediction, register renaming, instruction scheduling, and memory dependency prediction. [17]

2.6 gem5's O3 Core with FDP

The gem5 O3 core originally has five stages: fetch, decode, rename, IEW (issue, execute, writeback), and commit. The fetch issues requests to fetch the instructions, handles the branch prediction and dispatch instructions to the decode stage. After the instructions are decoded and passed to the rename stage, instructions are renamed to resolve false dependencies and stored in the ROB to keep track of the instructions. The IEW stage manages the out-of-order instruction scheduling using an IQ to track the instruction's readiness and a MDP to avoid memory order violations (read-after-write). The executed instructions are then retired in-order in the commit stage.

Our setup using FDP, additionally has the Branch and address calculation (BAC) stage, allowing the BPU to run ahead of the fetch stage and provide instructions to the fetch stage trough the FTQ.

Figure 2.1 provides an high-level overview of the resulting processor's pipeline.

The decoupled frontend implementation we use for this thesis has been implemented by D. Schall, available on GitHub [27].

[1] `https://www.gem5.org/`

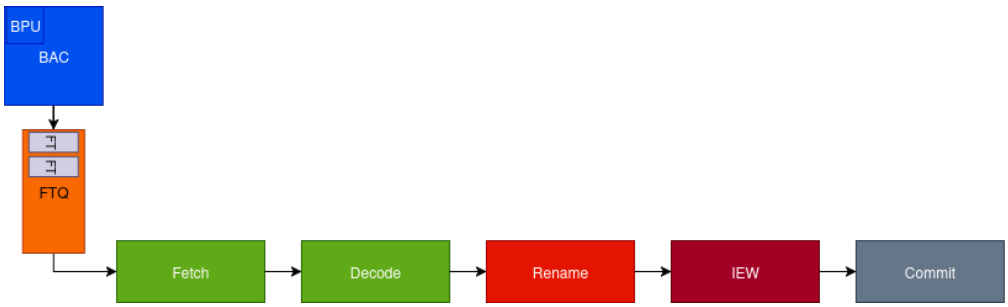


Figure 2.1: O3 Core pipeline with FDP

2.7 Multiple branch prediction

Assuming that 20% of the instructions are branches [29], the average basic block (equivalent to a fetch target in the context of FDP) size is 5 instructions. This means that performing one prediction per cycle is insufficient to fill an 8-wide pipeline, leaving 3 of 8 pipeline slots on average unutilized. This under-utilization creates a bottleneck that propagates through the entire pipeline, reducing the processor's ability to leverage instruction-level parallelism. This fetch bandwidth limitation becomes increasingly problematic as modern processors feature wider issue widths and deeper out-of-order execution capabilities that can theoretically handle much higher instruction throughput. Fetching more than one basic block per cycle is therefore necessary to get the most out of the core.

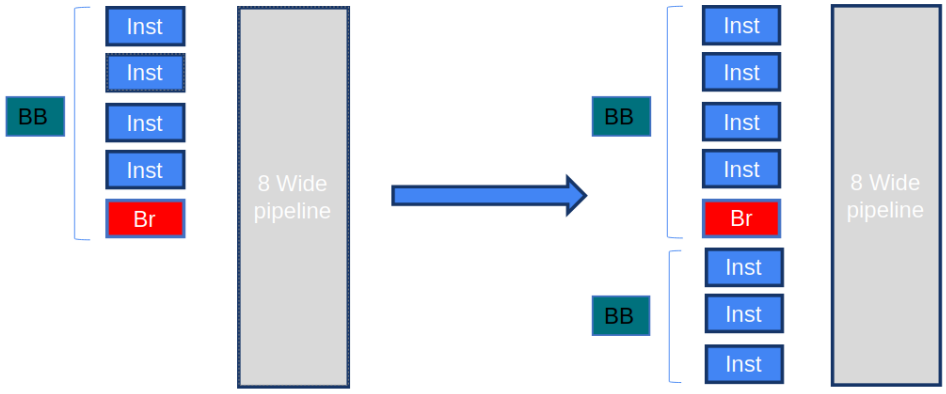


Figure 2.2: Two predictions per cycle for an 8-wide pipeline

Multiple branch prediction [32, 36, 40] is a technique to increase the instruction

fetch rate by predicting the targets of two or more branches per cycle and fetching the corresponding instructions. It usually relies on a dual(multi)-ported instruction cache and BTB. Figure 2.2 shows an example how two predictions per cycle enables utilization of the full pipeline's width.

2.8 Top-Down methodology

Modern processors incorporate hundreds of performance monitoring counters that track various aspects of execution, from cache misses and branch mispredictions to resource utilization and pipeline stalls. While this wealth of data provides detailed insights into processor behavior, it creates a significant challenge for performance analysis: developers and researchers often struggle to identify which metrics truly matter and which bottlenecks impact overall performance most. Traditional performance analysis approaches treat all performance issues as equally important, causing developers to pursue optimization efforts that yield marginal improvements to actual application performance. This scattered approach can result in wasted effort optimizing secondary bottlenecks while the primary performance limiters remain unaddressed.

A. Yasin [39] introduces the Top-Down methodology, it is a systematic performance analysis method specifically designed to address this challenge by providing a structured approach to bottleneck identification. Rather than overwhelming analysts with hundreds of individual counters, Top-Down creates a hierarchical classification system that organizes CPU pipeline slots into four main categories based on their fundamental impact on performance:

- Retiring: Slots used by operations that successfully complete and contribute to forward progress.
- Bad speculation: Slots wasted on incorrect speculation, including branch mispredictions and memory order violations.
- Frontend bound: Pipeline starved due to instruction fetch and decode limitations.
- Backend bound: Pipeline stalled due to execution resource limitations or data dependencies.

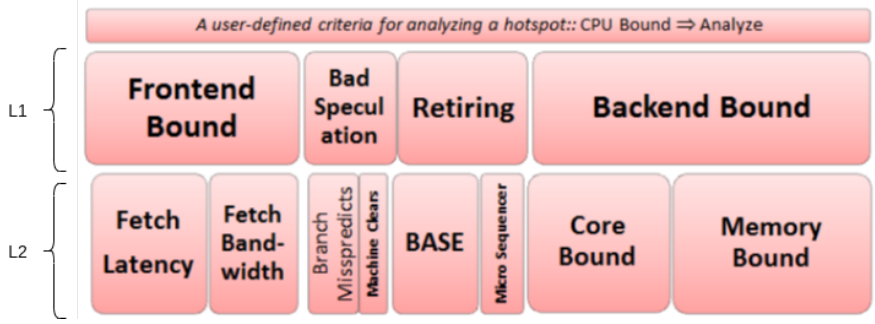


Figure 2.3: Top-Down Analysis Hierarchy L2 (taken from [39])

These four categories form the *Top Level (L1)* of the Top-Down hierarchy. Each L1 category can be further decomposed into *Level 2 (L2)* subcategories that provide more specific bottleneck identification. For instance, backend bound is subdivided into core bound and memory bound categories. Figure 2.3 illustrates the hierarchy structure through Level 2, while Levels 3 and 4 offer additional granularity for detailed performance analysis. Top-Down is a well-established analysis technique implemented in commercial processors, including Intel processors from Skylake onwards (accessible through Intel's VTune Profiler) and Arm's Neoverse processor family (supported by Arm's Telemetry framework).

The Top-Down methodology implementation in gem5 we use in this thesis was developed by Yaşar et al. [38].

3 Methodology

This Chapter describes the evaluation methodology and overviews some of the O3 core configurations we used for our experiments. We also give an overview of the hyperparameters we use to tune the predictors.

3.1 Benchmarks

For all the experiments evaluated, we exclusively used the *arm64* ISA available in gem5. The benchmarks were warmed up for at least 100M instructions, then run for at least another 200M instructions. We collected statistics from the *stats.txt* file generated by gem5 for each run. Table 3.1 gives an overview of the benchmarks we used. While SPEC benchmarks were simulated using system call emulation, the others were simulated using full system mode. We consider the differences between full system and system call emulation as not significant when it comes to benchmark results. SPEC workloads were simulated using SimPoints [21], a technique to speed up the simulation execution time for gem5 since they usually take quite a while to complete. Configuration details of the simulator are available on GitHub for server workloads [4] and SPEC workloads [5].

Benchmarks	Suite
nodeapp	Node.js web server [28]
libc, compression	Google's fleetbench [15, 28]
luindex, lusearch	Dacapo suite [7, 28]
gcc, mcf, xalancbmk, deepsjeng, leela	SPECrate 2017 Integer suite [34, 21]

Table 3.1: Benchmarks

3.2 Configurations

3.2.1 Configurations

The exact O3 core configuration used can be found on GitHub [6]. Table 3.2 gives an overview of the important O3 core parameters as well as cache hierarchy configurations.

Configuration name	w12	w36	bigger inst window
Pipeline width	12	36	64
Squash width	12	36	4608
BTB	32Ki entries	128Ki entries	256Ki entries
L1ICache	64KiB	256KiB	32MiB
L1DCache	64KiB	256KiB	32MiB
L2Cache	2MiB	8MiB	None
BPU	TAGE-SC-L 64KB	TAGE-SC-L 64KB	Infinite TAGE
MDP	Store sets	Store sets	Infinite PHAST
FTQ	50 FTs	150 FTs	400 FTs
IQ	600	1800	4800
ROB	576	1728	4608
LQ	200	600	1600
SQ	200	600	1600

Table 3.2: O3 Core configurations

3.2.2 Backend optimizations

The Tables 3.3 and 3.4 contain the hyperparameters we use in Chapter 5 to improve the accuracy of the branch predictor in the configuration *infinite TAGE* and memory dependency predictor in the configuration *infinite PHAST*.

TAGE logTagTableSize	20
TAGE shortTagsSize	20
TAGE longTagsSize	20
ITTAGE tagTableTagWidths	20
ITTAGE logTagTableSizes	20

Table 3.3: Infinite TAGE: TAGE-SC-L [31] and ITTAGE [30] hyperparameters

Number of rows	256
Associativity	16
Tag bits	20
maximum counter	100
LSQ DepCheckShift	0

Table 3.4: Infinite PHAST: PHAST [18] hyperparameters

4 Frontend

The very first bottleneck of a processor is its frontend, which acts as the crucial interface responsible for delivering instructions from memory to the execution engine. A fundamental principle in computer architecture is that a processor's execution speed for an application cannot exceed the rate at which it fetches its instructions [32]. This fundamental principle demonstrates that efficient instruction fetch mechanisms form the foundation of high-performance processor design. Fetching the correct instructions at the right time, fast enough to avoid backend starvation and continuously feed the execution units, has become an increasingly complex challenge over the years. As out-of-order execution techniques become more sophisticated and processors' issue windows grow larger, the demand placed on the frontend will intensify proportionally. The frontend will have to deliver instructions at higher rates.

This critical relationship between frontend performance and overall processor throughput motivates the first part of our work, improving the instruction fetch rate of gem5's O3 core. The gem5 simulator's O3 model provides an ideal platform for exploring these frontend dynamics due to its detailed modeling of modern superscalar processor architectures.

In this Chapter, we evaluate the effect of scaling and multiple branch prediction on the instruction fetch rate, highlighting the current bottlenecks of the O3 core's frontend. Our analysis includes an aggressive frontend design that we implemented within the gem5 O3 core, incorporating multiple branch prediction and an FTQ design able to substantially improve multiple branch prediction's performance. Furthermore, we consider fixing the frontend's bottleneck a fundamental prerequisite to illuminate the more subtle backend bottlenecks that we will analyze in subsequent Chapter of this work. Only by ensuring that the frontend can adequately supply instructions can we accurately assess the actual performance limitations of the execution backend. Also, we will focus our efforts on the BAC and fetch stages as the decode stage does not contain any critical or limiting components susceptible to result in a bottleneck for the frontend. Metrics of interest here are the instruction fetch rate, meaning the number of instructions fetched per cycle, and the Top-Down frontend bound [39], a metric quantifying the extent to which the frontend is a bottleneck for the core.

4.1 Simply scaling does not work

We start with the *w12* O3 Core configuration from Table 3.2 with FDP enabled. This *w12* configuration represents a conventional superscalar processor design. As a reminder, FDP is a proactive mechanism that hides the latency of instruction cache misses by issuing prefetches directly from fetch targets (FTs) addresses for the next cycles, thereby maintaining a steady flow of instructions.

Theoretically, increasing the fetch width—defined as the maximum number of instructions a processor is able to fetch per cycle—of the core is the most straightforward first step to take if we aim to increase instruction throughput, as otherwise the narrow fetch bandwidth might become a fundamental limiting factor that constrains overall performance. However, simply widening the fetch mechanism in isolation is insufficient. In doing so, we also have to proportionally scale up the entire *frontend infrastructure*, meaning not only the instruction cache capacity, but also the BTB size, the FTQ size, and various other supporting structures that collectively enable efficient instruction delivery. This comprehensive scaling approach leads us to the *w36* configuration detailed in Table 3.2.

Even though this *w36* configuration seems unrealistic compared to current commercial architectures in terms of resource allocation and complexity, this is entirely intentional as we orient our work toward a future direction, assuming that processors would evolve to reach such capabilities as manufacturing technology advances and design constraints relax. This scaling strategy also serves an important methodological purpose: it reduces the impact of other confounding variables like cache or BTB misses due to insufficient size on our analysis, allowing us to isolate and fully focus on the processor's fundamental behavioral characteristics and architectural bottlenecks.

This naive scaling approach, while logical in principle, unfortunately encounters a huge fundamental limitation: the inherent size distribution of the program's basic blocks, also referred to as fetch target in this architectural context. A basic block is a straight-line code sequence with no branches in except to the entry and no branches out except at the exit, its size is largely determined by the application's control flow patterns and compiler optimization strategies. The BAC stage will make a branch direction and target prediction (with the actual prediction logic delegated to the BPU) when it encounters a branch instruction, a BTB hit for the current program counter (PC). Upon making this prediction, the BAC creates the corresponding fetch target—essentially a continuous sequence of instructions extending from the starting PC to the branch's PC—and stores this fetch target in the FTQ for subsequent processing. This FT will then be consumed by the fetch stage during the next cycle, which issues a demand request for the cache line containing the instructions specified by the FT and stores the retrieved cache line in a dedicated fetch buffer for further processing. The predicted

branch target then becomes the starting PC for the subsequent fetch cycle, continuing this iterative process.

The problem emerges clearly: when the program's basic blocks are inherently too small, the available fetch width cannot be fully utilized during each cycle, inevitably leading to *fetch bubbles*. These bubbles represent wasted opportunities where some fetch slots remain unused due to frontend limitations. In the context of the Top-Down methodology [39] (Section 2.8), this inefficiency manifests as the frontend bound metric, indicating that the processor's performance is fundamentally limited by its ability to supply instructions rather than execute them.

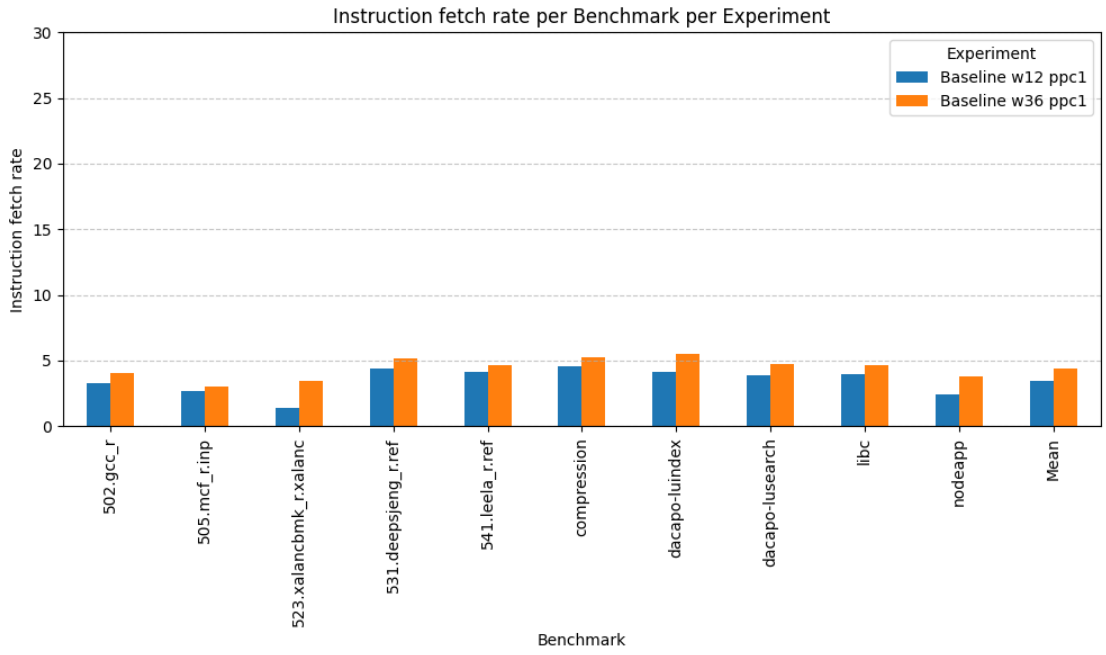


Figure 4.1: Instruction fetch rate - width 12 vs. 36

As the results shown in Figure 4.1 clearly demonstrate, the instruction fetch rate improvement is disproportionately small compared to the substantial increase in frontend resources, revealing diminishing returns from this scaling approach. The average value goes from 3.48 to 4.41. This observation highlights the fact that this traditional design is not scalable beyond certain limits, thereby demonstrating the critical need for a more sophisticated *scalable* frontend design that can overcome the basic block size limitation.

Additionally, Figure 4.2 shows the increase in the core's frontend bound, which means that more slots are idling, waiting for the frontend. For example, the value of 0.8

or 80% for the benchmark *libc* means that 80% of pipeline slots available for execution remain idle on average for the *w36* configuration, compared to around 50% for the *w12* configuration. This increase in frontend bound demonstrates that while the current frontend in the *w12* configuration already suffers from 50% of the slots idling, the problem becomes worse when the core's pipeline becomes wider (*w36* configuration), leading to increased under-utilization. This result provides additional motivation for our work, as there is significant room for improvement even for normal-sized processors.

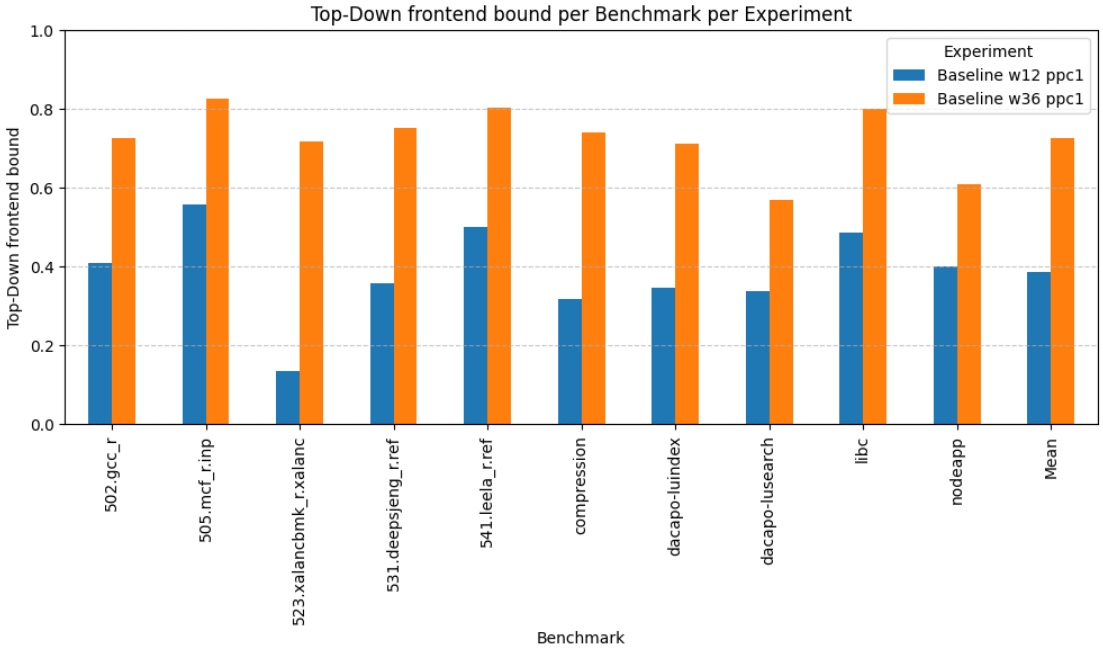


Figure 4.2: Top-Down frontend bound - width 12 vs. 36

Our analysis of the current frontend logic revealed two major bottlenecks:

1. Only one FT is consumed by the fetch stage per cycle, which is the main reason for the under-utilization of the fetch width.

2. Assuming a design able to consume multiple FTs per cycle, the fetch buffer would hinder it as soon as two FTs require access to different cache lines because in its current state, the fetch stage can only hold one cache line at a time.

To address both of these bottlenecks, we propose a frontend design leveraging multiple predictions per cycle together with a refactored FTQ that provides space for the cache

line of each FT directly. This combination is referred to as enhanced frontend in this thesis.

4.2 Enhanced frontend with aggressive FDP

In this section, we will first present the design of the enhanced frontend we implemented and then evaluate its performance on the benchmarks from Table 3.1.

4.2.1 Design

To address the first problem, we extended the BAC and fetch stages of the O3 core to both support multiple branch prediction. More precisely, this means that the BAC stage can produce multiple FTs per cycle while the fetch stage can consumes multiple FTs per cycle. The BAC relies on consecutive calls to the BPU to produce multiple FTs based on the previously predicted target. With this mechanism, we can predict a *path* of basic blocks in the program that is inserted into the FTQ. The FTs are inserted with the state *not ready* meaning the corresponding memory access has not been completed yet.

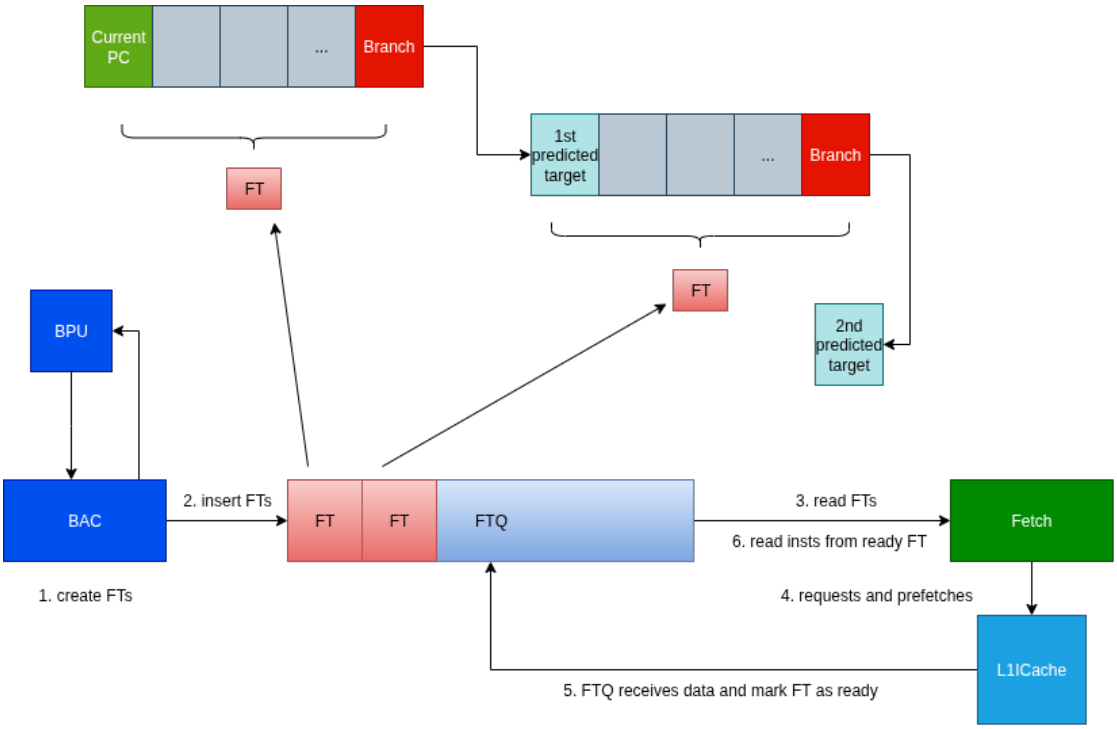


Figure 4.3: Enhanced frontend: Workflow's overview for two predictions per cycle

For the fetch stage to be able to consume multiple FTs per cycle, we modified the existing FTQ to store information (e.g., physical addresses after translation) on the state of the memory requests issued by the fetch stage at the fetch target granularity and handle the completion of these requests by storing the cache line locally in the corresponding FT. The FT is then marked as *ready*, and the fetch stage can start to read instructions from the FT's local fetch buffer. This solves the bottleneck of having a single fetch buffer, which was limiting the number of FTs the fetch stage could handle at once. Figure 4.3 gives an overview of the interaction between the two stages and the FTQ, successfully achieving multiple predictions per cycle.

Furthermore, this design is also beneficial for aggressive FDP as the requested instructions can be stored directly in the corresponding FT rather than sitting in the instruction cache and being requested again in the cycle when they are actually needed. This further reduces the latency of the fetch stage as it completely hides the instruction cache access latency. Prefetched instructions don't have to be requested again but are ready for subsequent cycles as soon as the prefetch completes.

This FTQ design results in additional physical memory needed for the entries. Assuming a fetch buffer of 64 bytes, this is 3.125KiB (50 entries) more for the *w12* configuration and 9.375KiB (150 entries) for the *w36* configuration. We consider this as relatively low, but there is also additional complexity for the implementation.

Multiple branch prediction, aggressive FDP and the refactored FTQ together build a solid frontend design that allows high instruction throughput by leveraging speculative execution to feed the backend. In the following, we will now evaluate its performance looking at the instruction fetch rate and the frontend bound.

4.2.2 Evaluation

To highlight the importance of the refactored FTQ and the impact of the fetch buffer bottleneck, we will first look at the effect of multiple branch prediction alone on the frontend's performance, then compare it to multiple branch prediction together with the refactored FTQ. This analysis is crucial for understanding whether our proposed frontend improvements can effectively mitigate the instruction supply bottlenecks that limit the performance gains typically expected from simply scaling processor resources, and to what extent these improvements justify the additional hardware complexity.

We consider the *w36* configuration and three multiple branch prediction scenarios:

- Two predictions per cycle (ppc2).
- Four predictions per cycle (ppc4).
- Six predictions per cycle (ppc6).

We will compare them to the performance of the baseline having one prediction per cycle (ppc1). We expect a positive correlation between the number of predictions per cycle and the fetch rate.

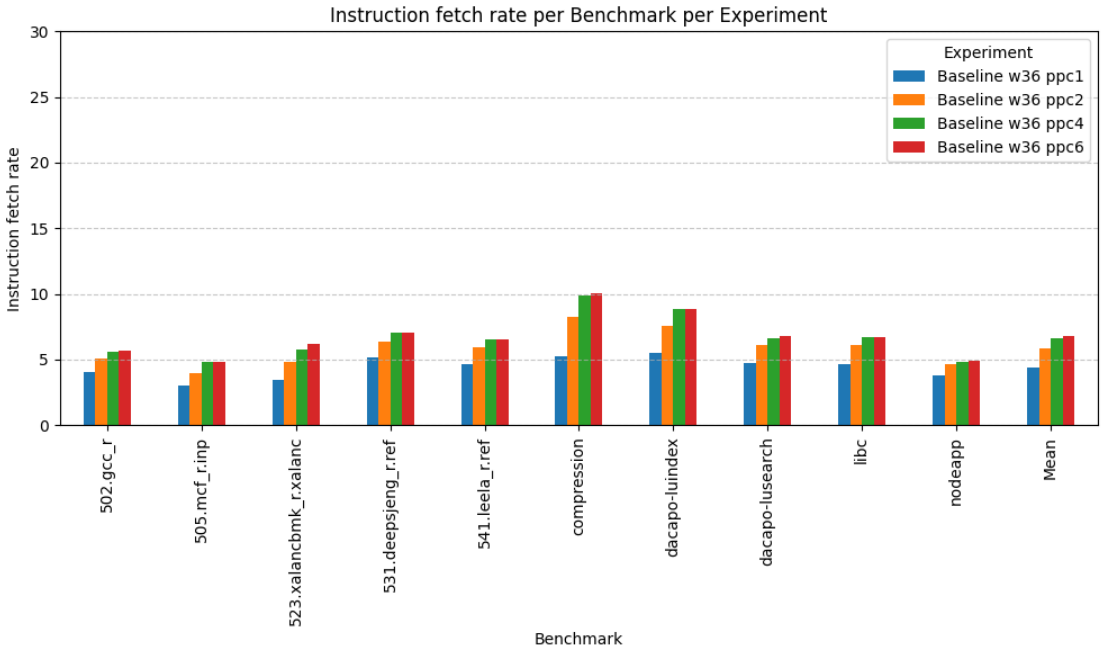


Figure 4.4: Instruction fetch rate: Baseline with multiple branch prediction

As Figure 4.4 shows, while there is indeed an improvement in instruction fetch rate with multiple branch prediction, the effects are limited. In particular, there seems to be no difference between four and six predictions per cycle. The average improves from 4.41 for one prediction to 6.76 for six predictions, meaning a speedup of 1.53X. This speedup does not reflect the expected impact of multiple branch prediction, hinting at a bottleneck. We attribute this to the fetch buffer bottleneck as explained in previous sections.

We now repeat the experiment with the same three scenarios, but this time with the enhanced frontend. The impact of the refactored FTQ is clear when looking at Figure 4.5: more predictions per cycle now effectively lead to a significantly higher fetch rate, with an outstanding speedup of 4.8X for *mcf* and on average a 3.4X for six predictions per cycle over one prediction. The average fetch rate for ppc6 reaches 15.2, and 4.42 for ppc1.

It is important to mention that the performance of the enhanced frontend and the baseline for one prediction are similar, 4.42 and 4.41, respectively. This similarity makes sense since the refactored FTQ's main advantage is removing the fetch buffer bottleneck when making multiple predictions per cycle. We explain the tiny improvement by the reduced latency enabled by the storage of prefetches directly in the FTQ.

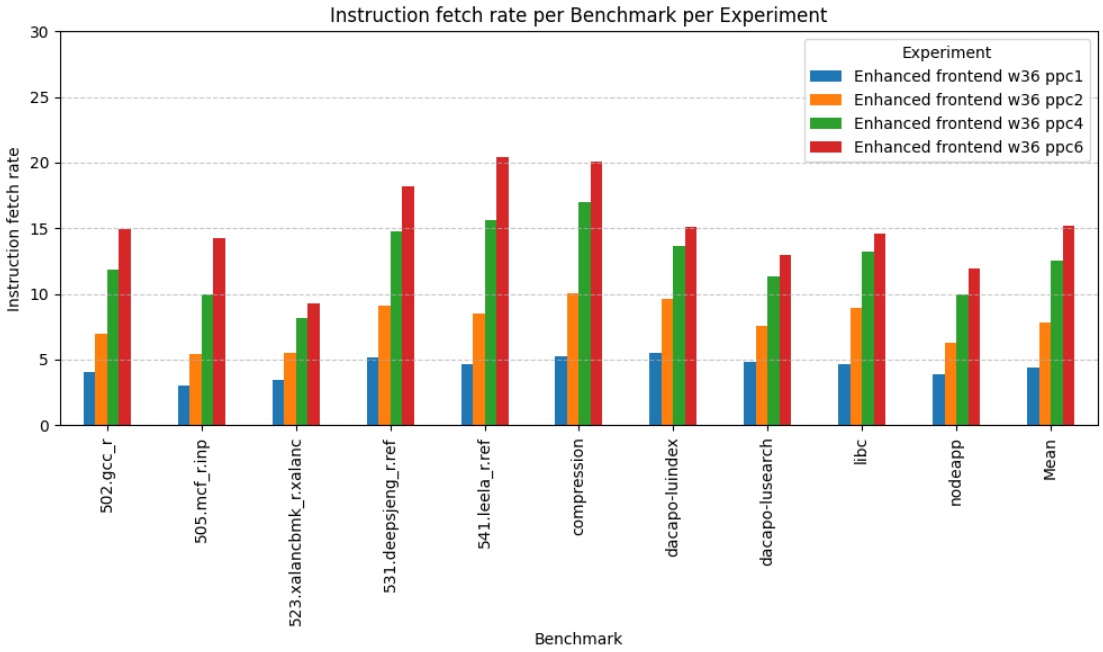


Figure 4.5: Instruction fetch rate: Enhanced frontend

This analysis demonstrates the significant impact of the fetch buffer bottleneck on multiple branch prediction performance and justifies the additional cost and complexity of implementing the refactored FTQ. A particularly compelling illustration is given by Figure 4.6, which shows the performance gap for six predictions per cycle when comparing architectures with and without the fetch buffer bottleneck. The average speedup is 2.3X, meaning the refactored FTQ design is more than twice as fast.

The difference is pronounced, revealing how the original fetch buffer design constrains the potential benefits of multiple branch prediction, while the enhanced frontend unlocks substantial performance improvements. Furthermore, these results conclusively show that multiple branch prediction can significantly improve the instruction fetch rate when properly supported by adequate infrastructure. This difference in performance validates our hypothesis that the fetch buffer represents a critical bottleneck in the O3 core's frontend.

Having demonstrated the substantial improvements in fetch rate achieved by the enhanced frontend, we now examine how these gains translate to overall frontend efficiency by analyzing the frontend bound.

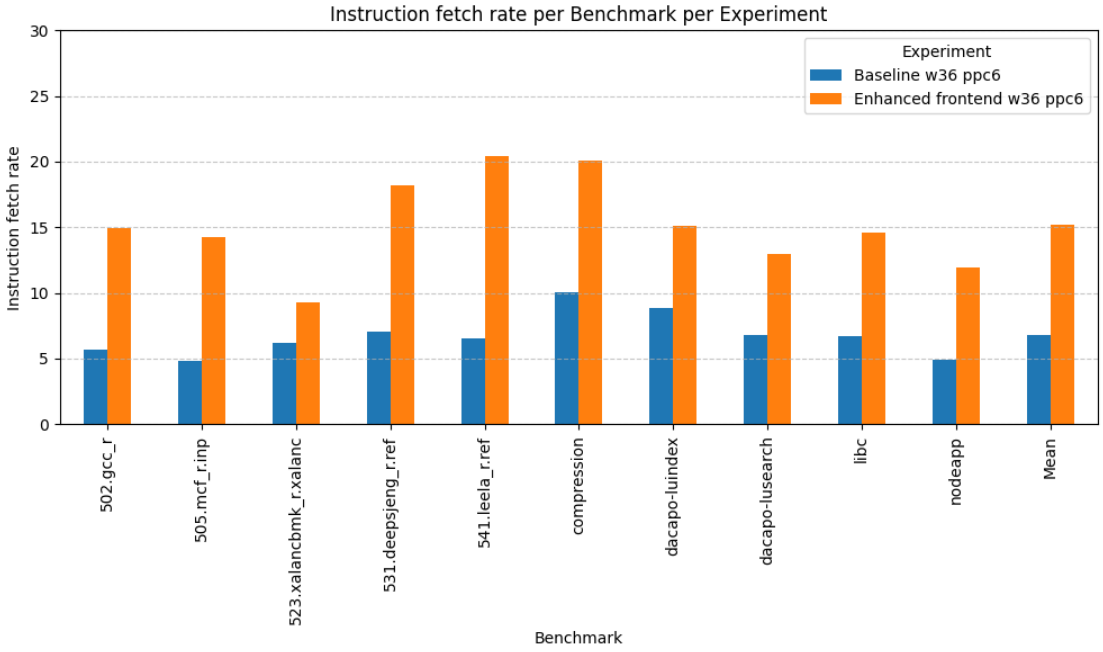


Figure 4.6: Instruction fetch rate: Performance of the refactored FTQ for six predictions

The substantial decrease in frontend bound for the enhanced frontend shown in Figure 4.7 demonstrates that our approach leads to better utilization of the fetch slots

with more predictions per cycle. A decrease of up to 96% is observable for *libc* when using six predictions per cycle over the one prediction, and on average 87.3%.

In comparison, multiple branch prediction without the refactored FTQ shows a decrease of up to 54.3% for *xalancbmk* using ppc6 over the one prediction, and on average 26%. Interestingly, in this scenario *libc* shows a decrease of only 11.5%. This lower frontend bound improvement was expected, as we demonstrated that the fetch buffer significantly limits multiple branch prediction performance.

Overall, this means that with the enhanced frontend, the core performance is no longer limited by its ability to fetch instructions. These results, combined with the fetch rate improvements, demonstrate that our setup can yield significant performance gains and improve bandwidth utilisation on a wide-pipeline architecture (*w36*).

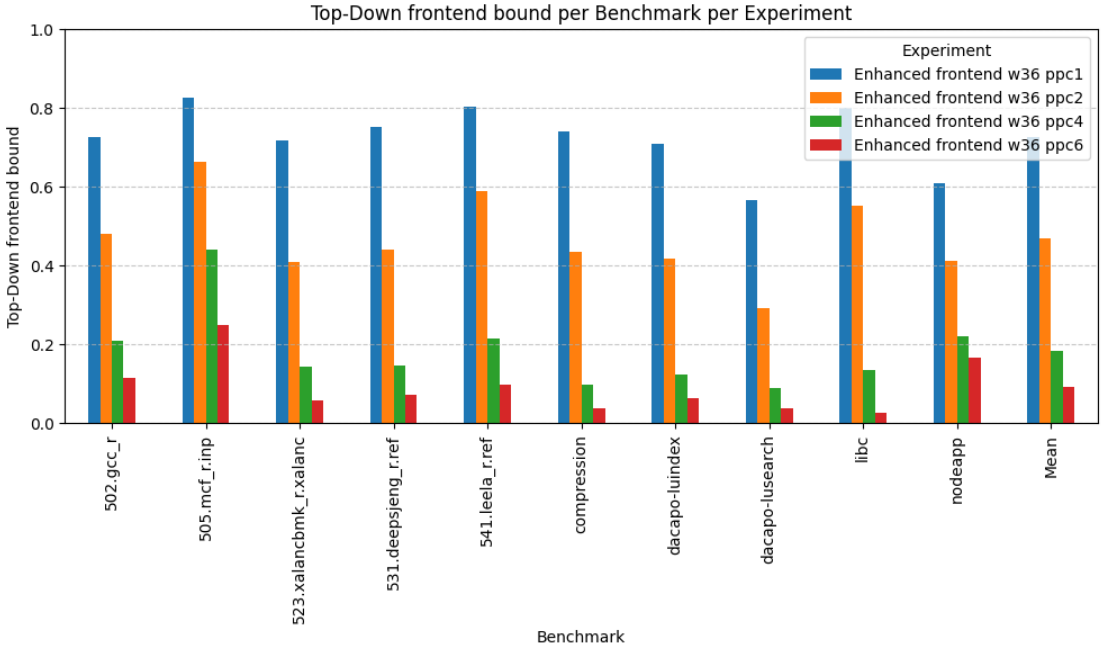


Figure 4.7: Frontend bound: Enhanced frontend

We also want to evaluate our enhanced frontend design's performance characteristics and effectiveness when operating under current resource constraints and limitations, specifically when configured according to our *w12* configuration. This analysis is critical as it interrogates the practical applicability of our approach in systems with more modest hardware budgets and existing infrastructure constraints. To maintain consistency with our previous evaluation methodology and enable direct comparison of results, we again consider the same four scenarios examined in the *w36* configuration analysis.

This approach allows us to isolate the impact of resource availability on our frontend improvements and determine whether the benefits observed in the *w36* environment translate effectively to more resource-constrained implementations. Additionally, we will not analyze the impact of the fetch buffer bottleneck on this configuration, as we expect a similar impact, and the effects have already been proven and discussed.

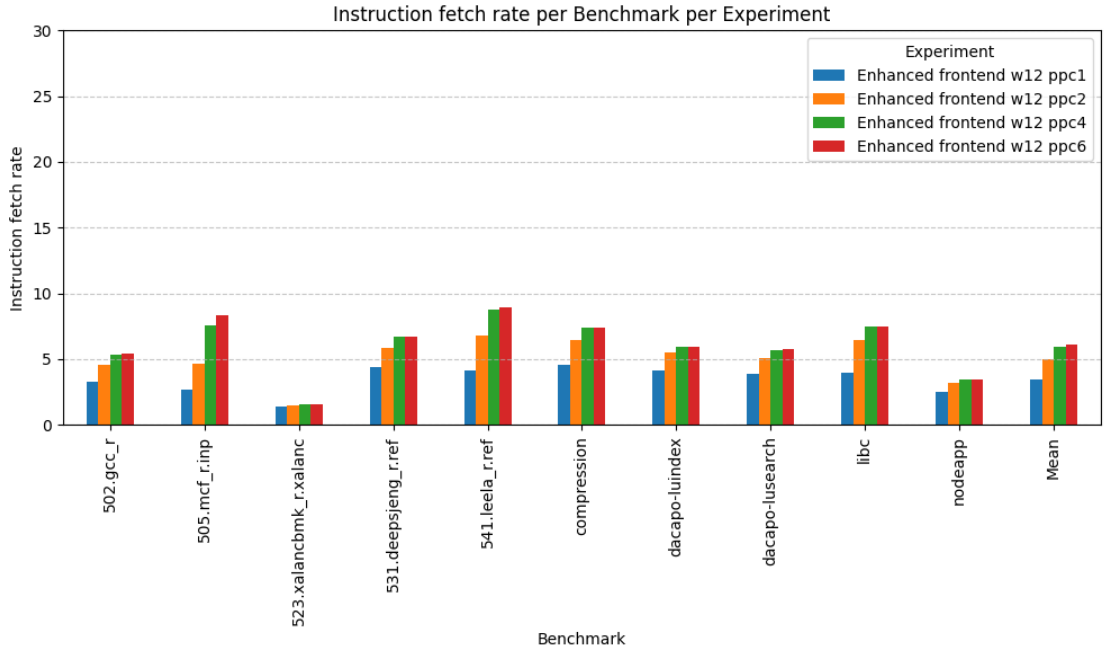


Figure 4.8: Fetch rate: enhanced frontend w12

As Figure 4.8 shows, there is an improvement in instruction fetch rate achieved by the enhanced frontend compared to the baseline. However, the improvements are less pronounced than those observed in the *w36* configuration. For one prediction per cycle, the average instruction fetch rate is 3.48, while six predictions per cycle achieves an average of 6.11. The maximum speedup from six predictions per cycle over one prediction per cycle is observed in *mcf* with 3.11×, whereas the average speedup is 1.74×. For comparison, the average speedup in the *w36* configuration is 3.4×, which means that the speedup is around 50% lower for *w12*. Moreover, *xalancbmk* exhibits a relatively low speedup of 1.16× for six predictions over one prediction, compared to 2.71× with *w36*. In resource-constrained environments, especially in the *w12* configuration, the limited fetch bandwidth and smaller buffer sizes may not provide sufficient headroom to fully exploit the advantages of the enhanced frontend. Compared to the *w36* configuration, the reduced effectiveness highlights the importance of having adequate resources to

support the enhanced frontend features.

Frontend bound also improves for the enhanced frontend compared to the baseline. The maximum decrease for six predictions per cycle over the baseline is achieved by *libc* with 99.7%, and the average is 86.1%. An interesting observation is that reducing the frontend bound works just as well on the *w12* configuration, in contrast to the lower instruction fetch rate improvement. This result makes sense as frontend bound expresses the utilization of the available fetch slots. In both configurations, multiple branch prediction is able to better utilize the available fetch width.

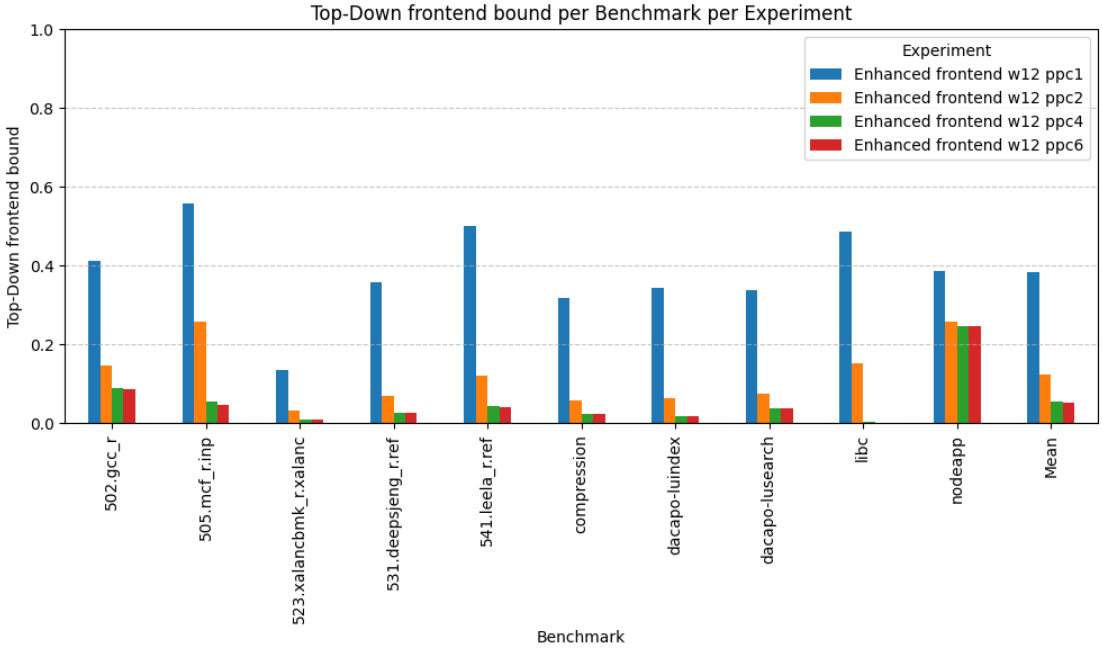


Figure 4.9: Frontend bound: enhanced frontend w12

Looking at both the *w36* and *w12* configuration results across our evaluation, we conclude that while the enhanced frontend design can indeed improve frontend performance when deployed on a standard configuration with limited resources, it demonstrates significantly greater effectiveness and is fundamentally complementary to the scaling approach. The enhanced frontend with the *w36* configuration consistently works better. It delivers more substantial performance improvements because more underutilized resources are available in the *w36* configuration, and our design can effectively leverage these additional resources to significantly improve the instruction fetch rate. While showing some improvements, the *w12* configuration is inherently limited by resource constraints that prevent the enhanced frontend from reaching its

full potential.

4.2.3 Path's prediction accuracy

As we use the BPU multiple times per cycle to predict branch targets and construct a path of basic blocks in the program flow, the cumulative prediction accuracy may be significantly lower than individual prediction accuracy. For example, assuming independent predictions and an accuracy of 0.97 (97%) for the branch predictor, the overall accuracy for a four-long prediction path is $(0.97)^4 = 0.88$ as we sequentially predict four branch targets. This degradation in prediction confidence can potentially lead to more harmful effects from mispredicted speculative execution, as incorrect early predictions can cascade through the entire prediction chain. However, this multiplicative effect is considerably harder to quantify accurately with advanced predictors like TAGE [33] because it is a sophisticated dynamic branch predictor that adapts based on program history, meaning the individual predictions are not truly independent and may exhibit correlations. Nevertheless, negative effects from reduced prediction accuracy are still observable in practice. We will demonstrate these effects in the next Chapter of this thesis, examining the backend performance implications and analyzing how frontend prediction quality impacts overall processor efficiency in the context of multiple branch prediction.

4.2.4 Real hardware opportunity

The focus of this study was to improve the frontend of the O3 Core for gem5 and evaluate the potential of multiple branch prediction; it was not to propose an implementation directly applicable to real hardware. However, there are opportunities for real processors to adopt a similar approach to ours, to substantially improve their instruction throughput and overall performance.

The refactored FTQ requires only low to moderate additional memory overhead depending on its size, making it a cost-effective enhancement that can be tailored to different processor designs. When used in conjunction with FDP, this FTQ design can not only effectively hide the instruction cache miss latency but also the access latency, making instruction supply considerably faster and more consistent across varying workload patterns.

Regarding multiple branch prediction, consecutive calls to the BPU within a single cycle might not be optimal for real hardware due to timing and power constraints. However, several promising approaches have already been presented and thoroughly evaluated by researchers to achieve up to three predictions per cycle [32, 36, 40], demonstrating the feasibility of this concept. In industry, AMD recently successfully

implemented a 2-ahead branch predictor (two predictions per cycle) in their Zen 5 processor family, achieving measurably improved performance in terms of both instruction fetch rate and IPC across a wide range of applications[11, 12].

These existing approaches and implementations could be strategically adapted to the architecture of a decoupled frontend and potentially extended to support more predictions per cycle as technology advances. While our evaluation was conducted in the gem5 simulation environment, the design principles and observed performance benefits translate well to real hardware implementations, particularly given the continuing evolution of processor design capabilities.

5 Backend

While the frontend operates in order, as demonstrated in previous sections, the out-of-order execution happens in the backend, which is responsible for leveraging the instruction-level parallelism present in the instruction stream provided by the frontend. With an aggressive frontend such as ours that can deliver substantial instruction throughput, the backend becomes the critical component that plays the most significant role in determining the resulting core performance and overall system efficiency. The backend must effectively manage the increased instruction flow, handle dependencies, and maximize parallel execution.

The goals of this chapter are as follows: First, it determines the theoretical performance upper limit of the O3 core. Second, it highlights the potential of multiple branch prediction for future processor design. Third, it brings to light critical bottlenecks and performance-limiting factors of the backend, which may also apply to real hardware implementations. Lastly, it provides insights for future research involving the gem5's O3 core.

We will first evaluate the effects of the enhanced frontend with multiple branch prediction on the backend and the overall performance, comparing the *w12* and *w36* configurations to highlight the performance gain from scaling. Then, taking the *w36* configuration with the enhanced frontend as a baseline for this Chapter, we will identify possible backend bottlenecks that may limit the performance improvement of multiple branch prediction and prevent the system from achieving optimal throughput. Finally, we will attempt to reduce their effects using optimization mechanisms such as improved branch prediction, improved memory dependency prediction, reduced misprediction penalty for the ROB to handle mispredictions more efficiently, and other targeted improvements.

5.1 Evaluation of the enhanced frontend's effects

Here is a quick overview of how the O3 core's backend works:

The reorder buffer (ROB) is a key structure as it buffers all the instructions waiting to be issued to an execution unit and the already executed instructions and their results, waiting to be retired in-order by the commit stage. The size of the ROB is therefore typically called the *instruction window*. The instructions are inserted into the ROB by

the rename stage, which renames the registers of the instructions beforehand, where needed, to resolve false dependencies and allow for more out-of-order execution. The instruction scheduler, referred to as the instruction queue (IQ), also stores instructions and schedules them. The IQ issues ready instructions to functional units where they are executed. An instruction is ready when it has all its operands available (for example, a value stored in memory requiring memory access introduces latency, results from earlier instructions mean it has to wait until those instructions are executed and retired). For the scheduling, the IQ relies on a dependency graph to track register dependencies and a memory dependency predictor to predict dependencies between load and store instructions and avoid memory order violations while allowing speculative execution.

Metrics of interest are:

- Cycles per instruction (CPI) which expresses how many cycles the core spends on each instruction on average. Here, lower is better. We show it in combination with Top-Down metrics as a CPI stack, showing the share of CPI belonging to each category.
- Instructions per cycle (IPC) is the inverse of CPI and expresses how many instructions are executed per cycle. IPC could be assimilated as *speed of execution* of the core. Minimizing CPI means maximizing IPC, thus maximizing the core's speed.
- Mispredictions per kilo instructions (MPKI), primary performance metric for the branch predictor unit. We display it as MPKI stack, showing the share of mispredictions due to the branch predictor and due to the BTB.
- Top-Down methodology [39] metrics. These metrics are helpful for bottleneck identification.
- ROB occupancy showing how many instructions are available to the backend.

We previously showed that our enhanced frontend improves the instruction fetch rate, so more instructions are fetched and streamed to the backend. However, that does not directly mean that the backend actually executes more instructions or that overall performance will necessarily improve proportionally. We need to look closely at the backend mechanisms to understand what happens.

The first interesting effect is how the number of instructions in the ROB changes. We can visualize the distribution of the ROB occupancy over the simulation for our four multiple branch prediction scenarios. Figure 5.1 for the benchmark *libc* clearly demonstrates how the frontend fills the ROB faster for more predictions per cycle.

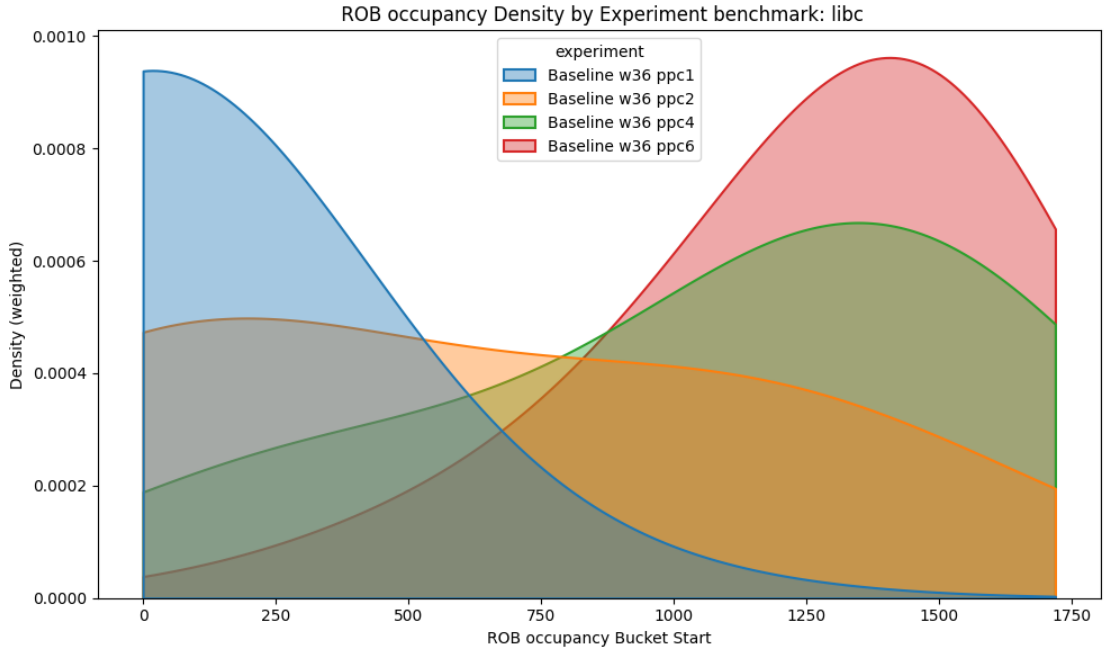


Figure 5.1: ROB occupancy for benchmark libc

This increase in ROB occupancy is expected and necessary to leverage more ILP in the program. Figure 5.2 shows the mean of the ROB occupancy, giving an overview of the effect on the ROB for all the benchmarks. The growing occupancy percentage from the ROB capacity across the benchmark suite confirms the trend we observed for *libc*.

Multiple factors can explain variations in the intensity of the effects:

- The instruction fetch rate: If the fetch rate improvement was already limited, the ROB can obviously not fill faster.
- Renaming registers: For programs relying heavily on the same type of registers (for example, integer registers), if none are available, instructions have to wait in the rename stage until some are freed, meaning the completion or squashing of previously renamed instructions.
- MPKI: Even with a high instruction fetch rate, if the instructions are on the wrong prediction path, they will be squashed once the branch is resolved. Benchmarks with a high MPKI, therefore, typically have a lower ROB occupancy because a significant part of the speculatively fetched instructions are discarded. *mcf* is a good example of this. With around 19.7 MPKI, it exhibits the smallest ROB

occupancy improvement. As we will demonstrate later, this lower ROB occupancy is not the only drawback of a high MPKI.

- Serializing stalls: Some programs rely on so-called *serializing instructions* that force the processor to complete all prior instructions before starting any subsequent instructions. A serializing instruction could be, for example, to ensure a precise architectural state before continuing the execution. In our model, these instructions cause the rename stage to stall until the ROB is fully drained, preventing more instructions from being inserted in ROB and executed.

  With a serializing stall cycles rate [1] between 20 and 23% (for ppc1, 2, 4, and 6), *nodeapp* is a good example of how this impacts ROB occupancy.

  Furthermore, serializing stalls can stall the stages before rename (BAC, fetch, and decode), reducing the instruction fetch rate.

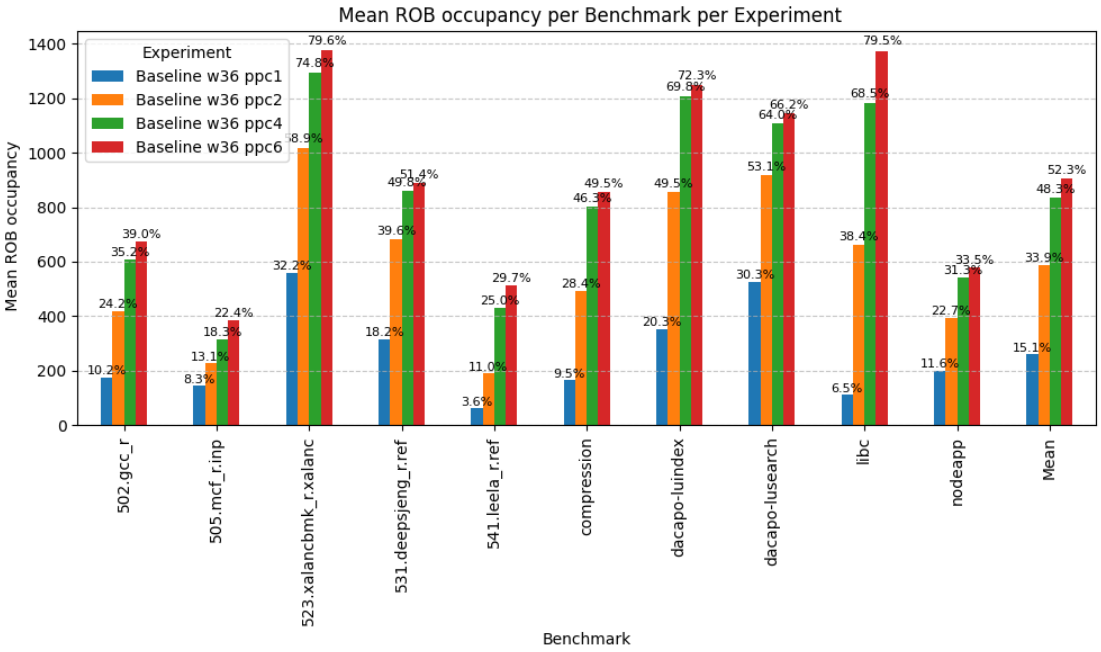


Figure 5.2: Backend baseline - Mean ROB occupancy

Combinations of these factors are also possible and might further decrease the effect of multiple predictions per cycle on ROB occupancy. It is evident that if the ROB occupancy does not increase, overall performance will not improve either.

[1] number of serializing stall cycles divided by total number of cycles during the simulation

Figure 5.3 shows the core's IPC on the different benchmarks for *w36* and *w12*. It is clear that there is already some performance improvement from scaling. The average IPC for *w12* goes from 2.31 for one prediction to 2.57 for six predictions, and from 2.97 to 3.88 for *w36*, respectively.

Focusing on *w36* from now on since it is the baseline for this chapter, an overall observation is that more predictions per cycle actually lead to worse performance for some benchmarks, as evidenced by the decreasing IPC values. *Compression* shows a speedup of 0.82X (a speedup lower than one actually means a decrease in performance), and the average speedup is 1.2X. This counterintuitive result suggests that simply providing more instructions to the backend does not automatically translate to better performance when other bottlenecks are present, it may actually hurt performance. Interestingly, *libc* performs quite well, going from 4.57 for one prediction per cycle to 13.46 for six predictions per cycle, achieving a substantial speedup of 2.9X.

Nevertheless, this raises critical questions: Why do not the other benchmarks improve to a similar extent? What specific bottlenecks prevent them from realizing similar performance gains? Where exactly do these limitations manifest in the processor pipeline? We will address these questions in the next section of our work.

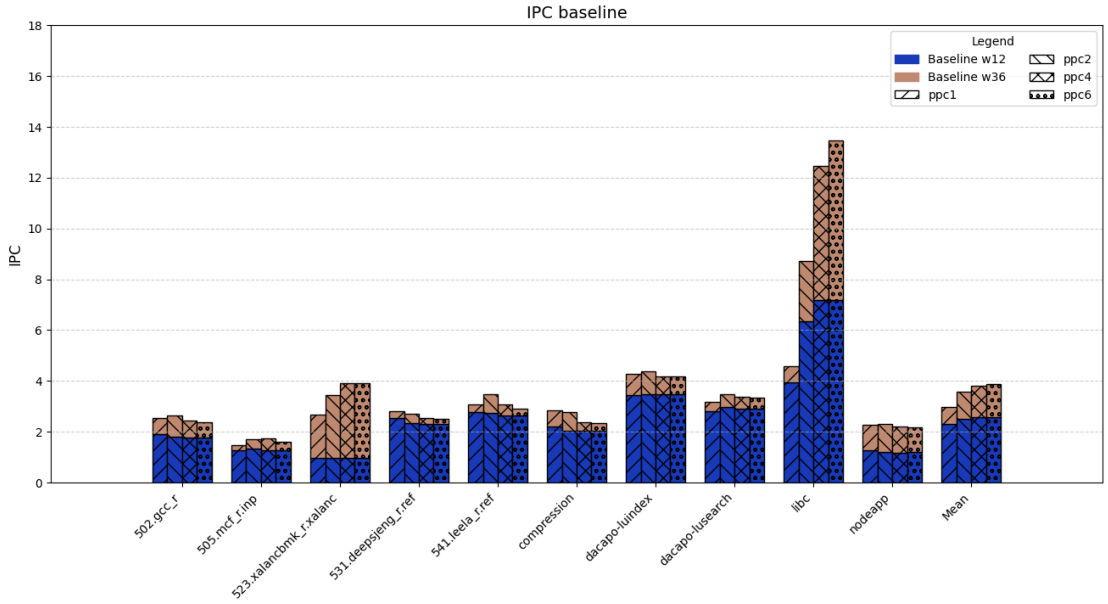


Figure 5.3: Backend baseline IPC

5.2 Bottlenecks

As we showed in the previous section, the increase in instruction throughput does not automatically translate to performance improvement. We explained that resource availability (registers, instruction window size) and serializing instructions are the first bottlenecks to frontend performance. While resource limitations can be purely and simply solved by scaling, serializing instructions are part of the program's logic and cannot be modified by the core. However, as Figure 5.2 showed, the ROB is able to fill faster for most of the benchmarks even with the current resources. We therefore choose not to focus on the renaming limitations but rather on the behavior of the rest of the pipeline as a first step.

CPI stacks enable us to visualize the proportion of each Top-Down metric in the CPI that the processor needed for the program. We use CPI stacks showing L1 metrics: retiring, frontend bound, backend bound and bad speculation. They are referred to as L1 CPI stacks.

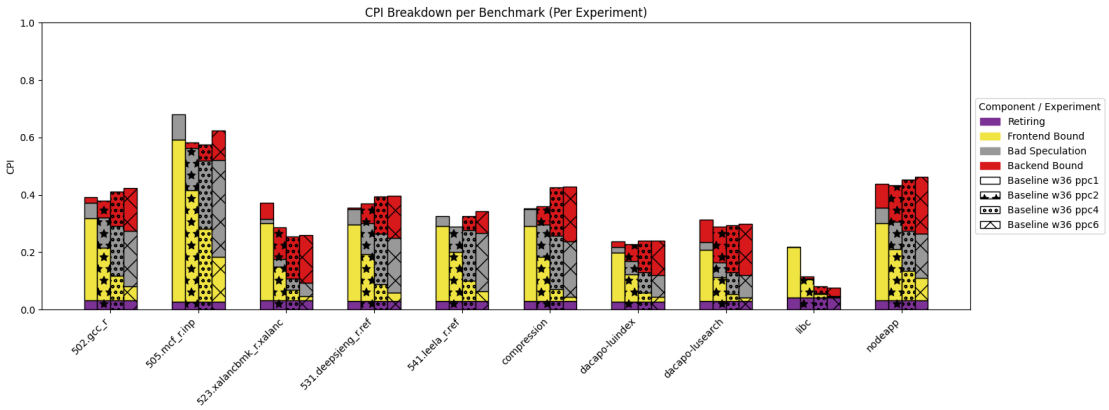


Figure 5.4: Backend baseline - L1 CPI stacks

As Figure 5.4 shows, the proportion of frontend bound consistently diminishes for more predictions per cycle, as we previously demonstrated through our frontend analysis. The core's performance is no longer bottlenecked by its capacity to fetch instructions.

However, simultaneously, bad speculation and backend bound grow, eventually exceeding the frontend bound decrease and ultimately deteriorating the CPI. To better understand what happens, we extend L1 CPI stacks to L2 CPI stacks, giving a detailed view of the backend bound and bad speculation metrics. L2 of backend bound includes memory bound and core bound, and L2 of bad speculation includes branch mispredicts and machine clears (in our study, memory order violations). We modified the L2

Top-Down methodology metrics to also include serializing stalls as part of the backend bound, since they create pipeline stalls that prevent the backend from making progress (backend bubbles).

Looking at Figure 5.5, we noticed three metrics significantly growing with more predictions per cycle for the benchmarks exhibiting a performance decrease: BadSpec memOrderViolations, BadSpec mispredicts, and backend memoryBound. We will therefore focus on these in our analysis .

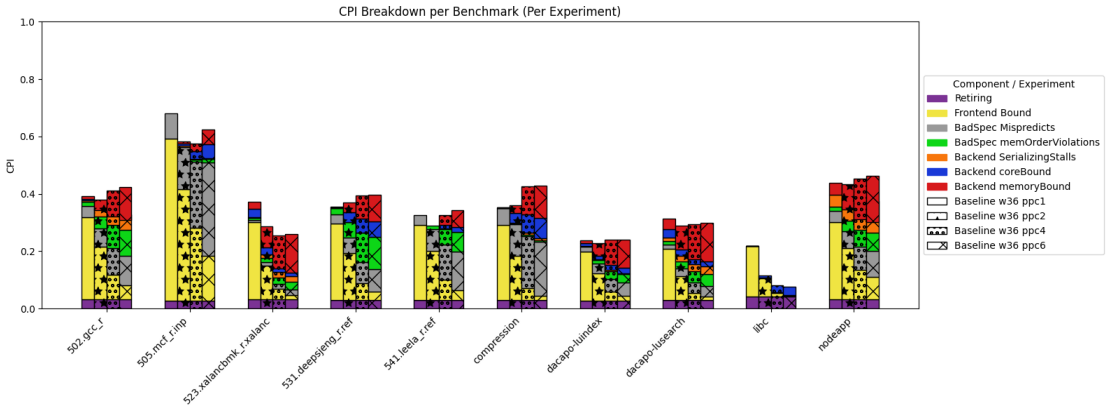


Figure 5.5: Backend baseline - L2 CPI stacks

5.2.1 Bad speculation

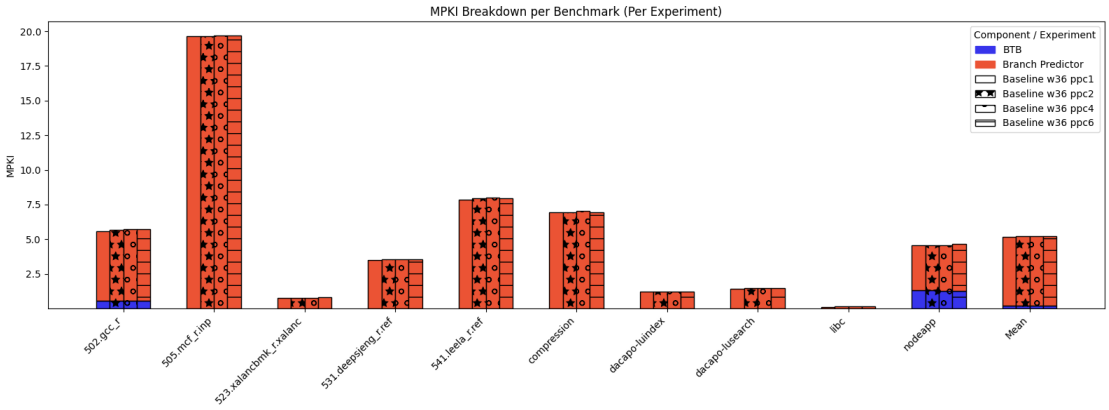


Figure 5.6: Backend baseline - MPKI stacks

As observed in Figure 5.4, the CPI does not decrease with multiple predictions per cycle for most of the benchmarks, and in some cases it grows, meaning the overall performance is getting worse with more predictions. Multiple branch prediction allows for more speculative execution multiple blocks ahead compared to a single prediction scheme. As we saw, it fills the ROB faster, giving more room for parallel execution. Counterintuitively, this is actually the root cause of the performance drop observed. To understand this, we must consider what happens when the processor has to squash some instructions from the ROB. The squash width of the O3 core defines how many instructions the ROB can squash in one cycle.

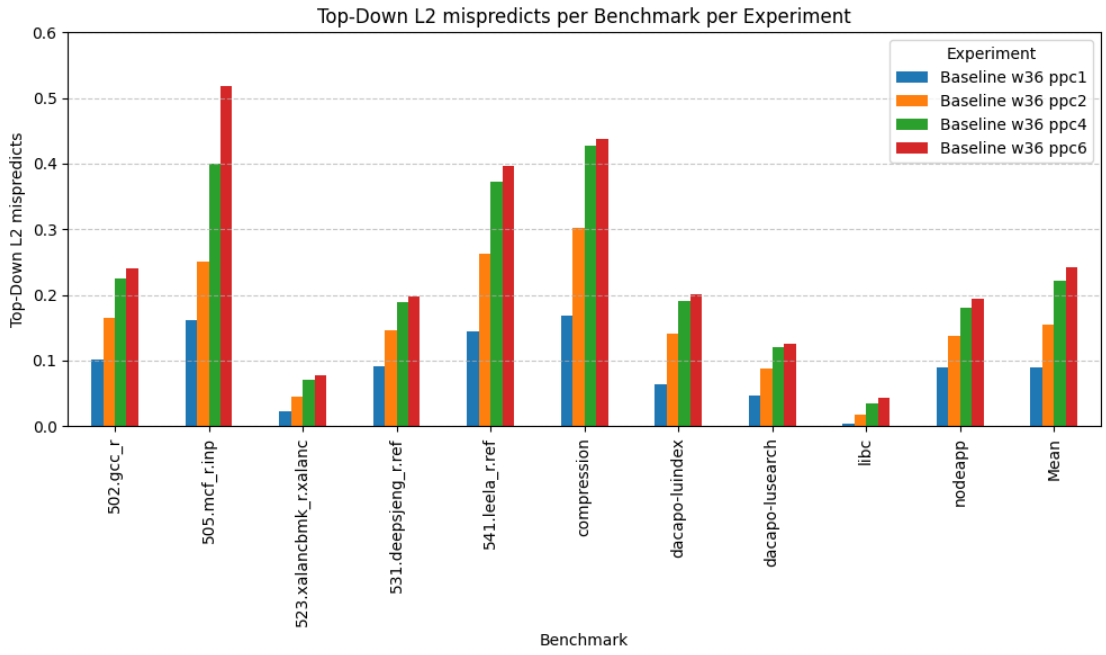


Figure 5.7: Backend baseline - Top-Down L2 mispredicts

We compare one prediction (ppc1) vs four predictions (ppc4) per cycle in the worst case scenario where the first prediction is wrong: For ppc4, the frontend is much more aggressive, filling the ROB with many more instructions than ppc1, and these instructions will all need to be squashed before the backend can accept any further instructions. Therefore, the ROB will need more cycles to squash the instructions from the wrong path, meaning that ppc4 leads to more recovery bubbles (cycles where no new instructions can be processed) than ppc1, assuming a fixed squash width. Even if the performance of the branch prediction unit does not directly degrade with more predictions per cycle (meaning MPKI does not significantly increase), as Figure 5.6

shows, the penalty for the ROB still increases substantially.

This increase is demonstrated by Figure 5.7, which shows the L2 mispredicts bound of the processor. A benchmark like *mcf*, which already has quite a high MPKI, sees its misprediction penalty increasing significantly, meaning that more and more cycles are wasted on recovering from mispredictions. These cycles account for more than 50% of the cycles for ppc6. Benchmarks with a smaller MPKI also show a lower increase in mispredicts bound, demonstrating that the accuracy of the BPU matters.

Similarly, with more instructions being scheduled simultaneously for execution, a poor performance of the memory dependency prediction unit may fail to detect memory dependencies, leading to more memory order violations than with a single prediction scheme. When the memory dependency predictor incorrectly allows a load instruction to execute before a store that it actually depends on, a memory order violation occurs, requiring the processor to recover by squashing the incorrectly executed instructions and their dependents. This increase in memory order violations means that the ROB has to squash more often and potentially more instructions as the number of predictions per cycle increases.

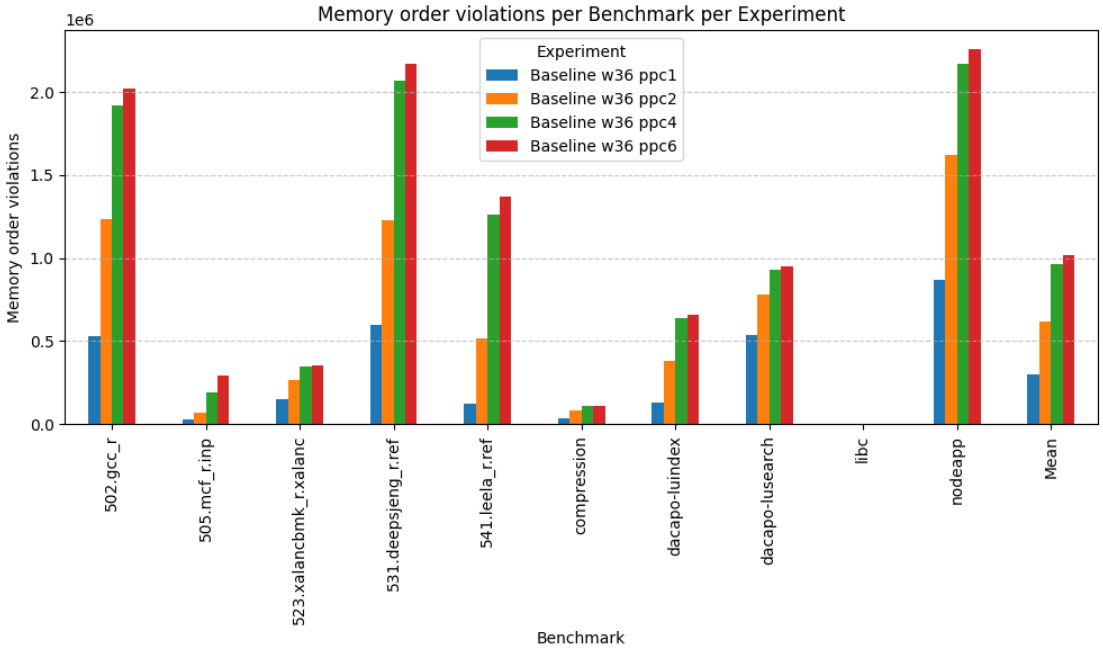


Figure 5.8: Backend baseline - Memory order violations

Figure 5.8 shows how memory order violations increase with more predictions per cycle, confirming our explanation. Top-Down L2 memOrderViolations shows a similar

growth, as shown in Figure 5.9, meaning that the processor indeed spends a bigger share of its cycles squashing due to memory order violations.

This analysis explains the counterintuitive adverse effect of multiple branch prediction on performance observed for some benchmarks, as well as the consistently increasing share of CPI belonging to L1 bad speculation (which comprises L2 mispredicts + memOrderViolations). The fundamental issue is that while multiple predictions per cycle successfully increase instruction fetch rate and fill the processor's instruction window more aggressively, they also amplify the recovery costs when speculation fails. When branch predictions are incorrect or memory dependencies are violated, the processor must squash a much larger number of instructions compared to a single prediction approach.

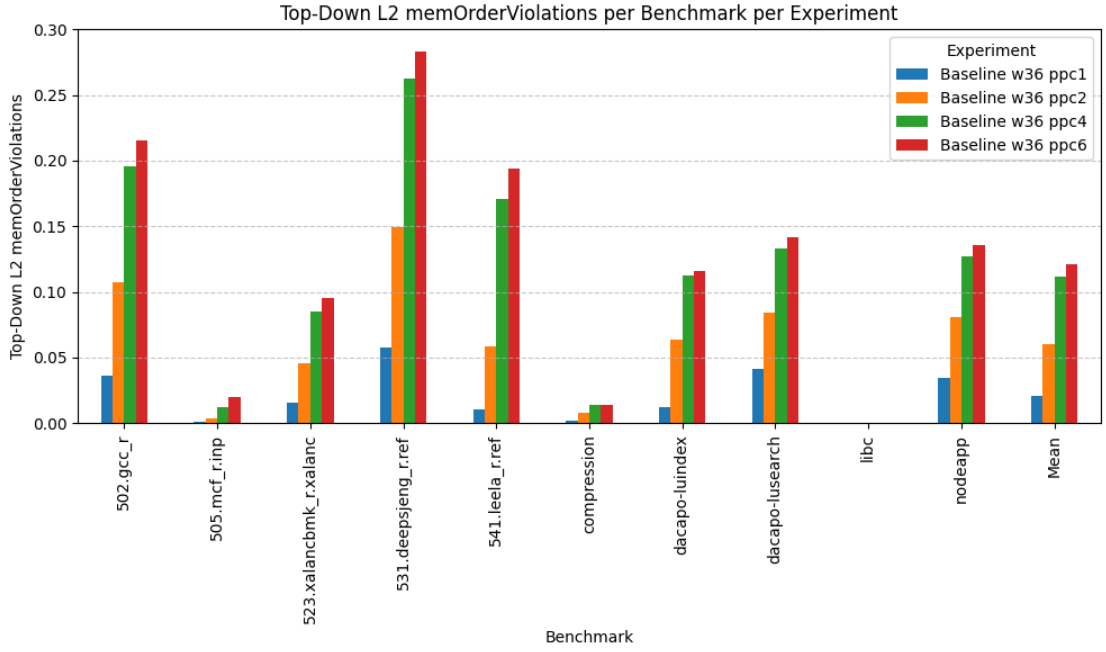


Figure 5.9: Backend baseline - Top-Down L2 memOrderViolations

Figure 5.10 shows the number of cycles the ROB spends squashing. The trend is clear and confirms our hypothesis: Multiple predictions per cycle increase the recovery penalty due to the increasing number of cycles needed for the ROB to squash.

The data clearly demonstrates that for benchmarks with inherently high misprediction rates (*mcf*, *compression*) or complex memory access patterns (*deepsjeng*, *nodeapp*), the recovery penalty increases and becomes the dominant factor limiting performance, effectively negating any gains from the enhanced frontend's improved fetch capabili-

ties. This observation explains why some benchmarks perform worse with multiple predictions per cycle despite the theoretical advantages of more aggressive speculation. Moreover, it also explains why *libc* and *xalancbmk*, which exhibit both low MPKI and few memory ordering violations, are able to benefit from multiple predictions per cycle since the ROB penalty does not increase as much.

We identified three bottlenecks related to bad speculation that can be optimized:

- ROB squash penalty: This is the number of cycles the pipeline cannot make progress because the ROB is squashing. Lowering this penalty means lowering the bad speculation bound.
- BPU accuracy: A more accurate BPU would lower the number of squashes due to mispredictions.
- MDP accuracy: A more accurate MDP would lower the number of squashes due to memory order violations.

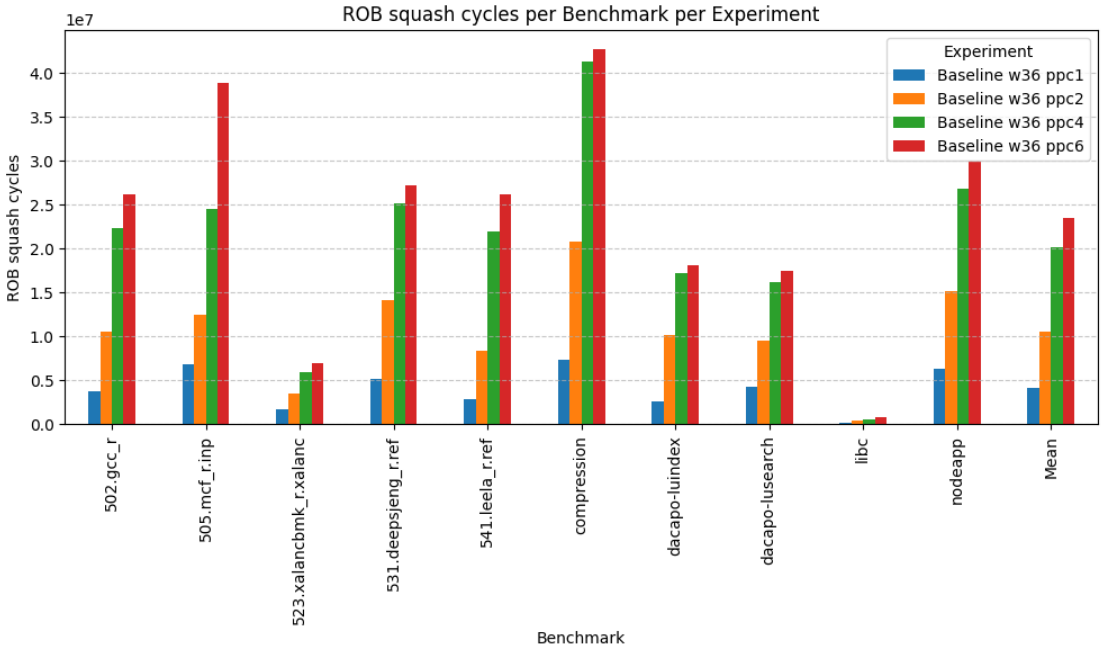


Figure 5.10: ROB squash cycles: Backend baseline

While speculative execution represents one dimension of the performance bottlenecks, memory bound constitutes another critical factor. Our analysis now shifts to examining how memory bottlenecks interact with multiple branch predictions per cycle.

5.2.2 Data cache

Cache misses can severely impact backend performance, with increasingly high penalties depending on the cache miss level (L1, L2, or main memory). These penalties can significantly affect the efficiency of out-of-order execution, particularly in scenarios where multiple instructions are dependent on a load instruction that experiences a cache miss. The straightforward and immediate drawback is that all the dependent instructions in the dependency chain must wait dozens or even hundreds of cycles to be ready to execute, even though execution units remain available and could theoretically process other work. One key idea of out-of-order execution is to mask such a miss latency by executing other ready instructions. The higher instruction throughput and the larger instruction window should give the core more flexibility to hide the latency. However, there are two limitations to this. First, latency hiding is limited for a workload with long instruction chains, depending on a memory access, because the instructions are not ready. Second, for workloads with complex and frequent memory accesses, if more instructions are in the instruction window but they mostly trigger cache misses, the latency hiding mechanism of out-of-order execution becomes ineffective.

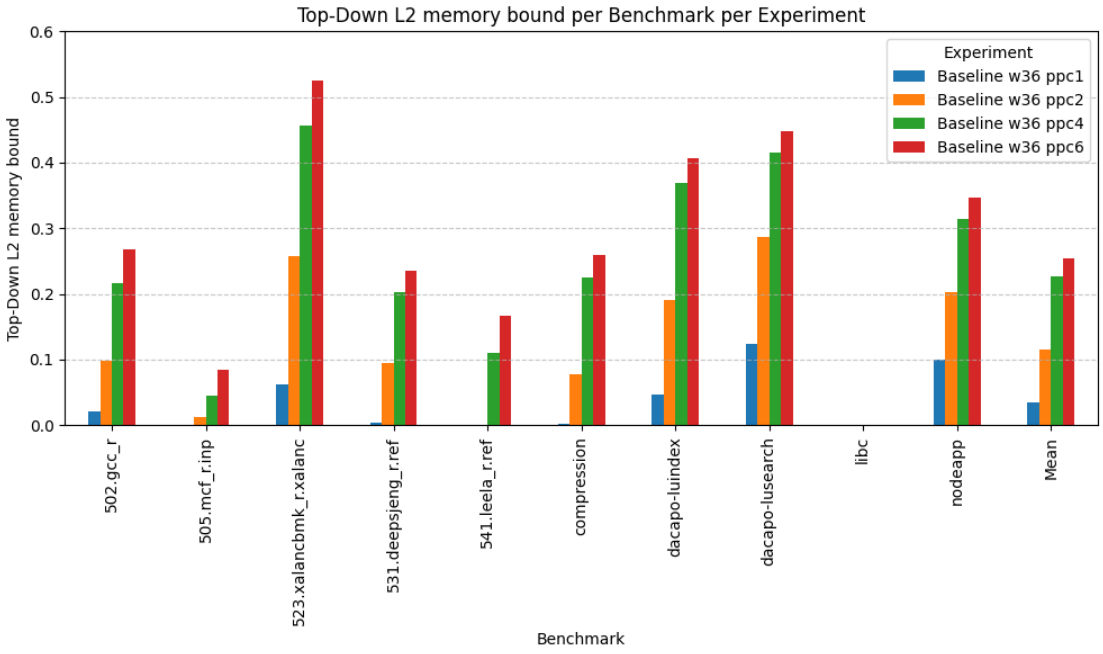


Figure 5.11: Backend baseline - Top-Down L2 Memory Bound

With an increasing throughput of instructions from the frontend, even though the

total number of cache misses might not go up (same workload, instructions just streamed faster), performance can stagnate due to long dependency chains or an increasing number of simultaneous cache misses.

However, unlike bad speculation bottlenecks, it should not decrease the core's performance.

Figure 5.11 shows the increase of L2 memory bound with more predictions per cycle. This result confirms our hypothesis of the increase in execution slots remaining unused due to cache misses.

We therefore identify the data cache as a bottleneck when the instruction throughput increases.

5.3 Solutions and evaluation

Now that the main bottlenecks limiting multiple branch prediction performance are clearly identified, we will attempt to solve them one after the other, building step by step an optimized backend able to handle the increasing instruction throughput pressure caused by multiple predictions per cycle efficiently. We will start by looking at the squash penalty, as we believe it is one of the most important factors.

5.3.1 The one cycle squash experiment

As explained previously, multiple predictions per cycle fill the ROB faster. Therefore, it takes longer to drain in case of wrong speculative execution. The solution we propose is *one cycle squash (OCS)*, a configuration aiming to minimize the ROB recovery penalty. Minimizing the recovery penalty amounts to setting the core's squash width equal to the instruction window, the ROB size. By doing so, the ROB can theoretically squash its full capacity in one cycle, effectively removing the bottleneck due to the increasing number of cycles needed to discard instructions from the wrong path.

Figure 5.12 compares the number of cycles the ROB spends squashing between the baseline and *one cycle squash*. ROB squash cycles remain significantly lower for *one cycle squash* than for the baseline, with a negligible increase for more predictions per cycle. The increase can be explained by the fact that more speculative execution may lead to more squashes, for example, the increasing number of memory order violations for more predictions as shown in Figure 5.8. The impact of this configuration is clear: the ROB's squashing penalty does not dramatically increase anymore. This optimization should significantly reduce the observed bad speculation bottleneck.

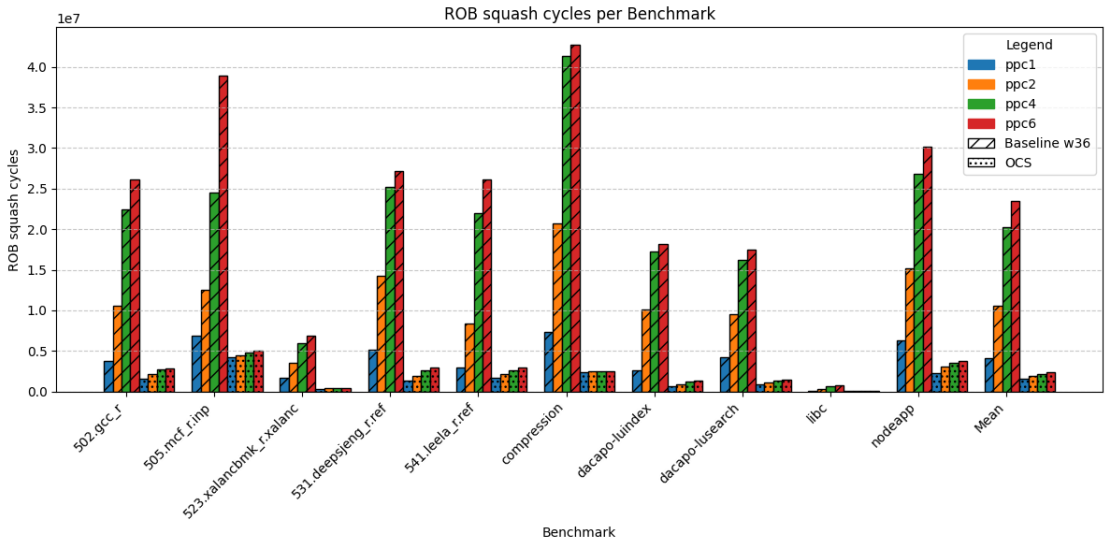


Figure 5.12: ROB squash cycles: Baseline vs one cycle squash

Figure 5.13 shows the IPC for *one cycle squash*. The first and most likely the most important observation is that all the benchmarks exhibit a positive impact of multiple branch prediction on performance, which is a great improvement compared to the baseline, where some benchmarks suffered from a decrease in performance. This result also confirms our hypothesis on the importance of the ROB squash penalty and its impact on performance with multiple predictions per cycle. The average IPC is 3.02 for one prediction and 4.47 for six predictions. The lowest speedup from six predictions per cycle over one prediction is 1.06X achieved by *nodeapp*, the highest is again *libc* with 2.96X, and the average is 1.39X. As a reminder, the average speedup for the baseline was 1.2X, meaning a 15.8% increase in the average speedup.

The disappearance of the negative effect on performance, the significant increase in IPC, and the increased average speedup over the baseline demonstrate that this bottleneck needs to be addressed when using multiple branch prediction, and leave us with this key *takeaway*: Being able to fill the ROB faster to allow more speculative execution requires being able to empty it faster when the speculation turns out to be wrong, otherwise, the recovery penalty grows and hurts the overall performance, leading to diminished IPC.

Reducing the bad speculation penalty is helpful for recovering from wrong speculative execution faster. However, to reach optimal performance, there should not be any wrong speculative execution at all. This observation leads us to the next part of our bottleneck-solving approach: investigating the effects of improved branch prediction

accuracy and a more accurate memory dependency prediction unit.

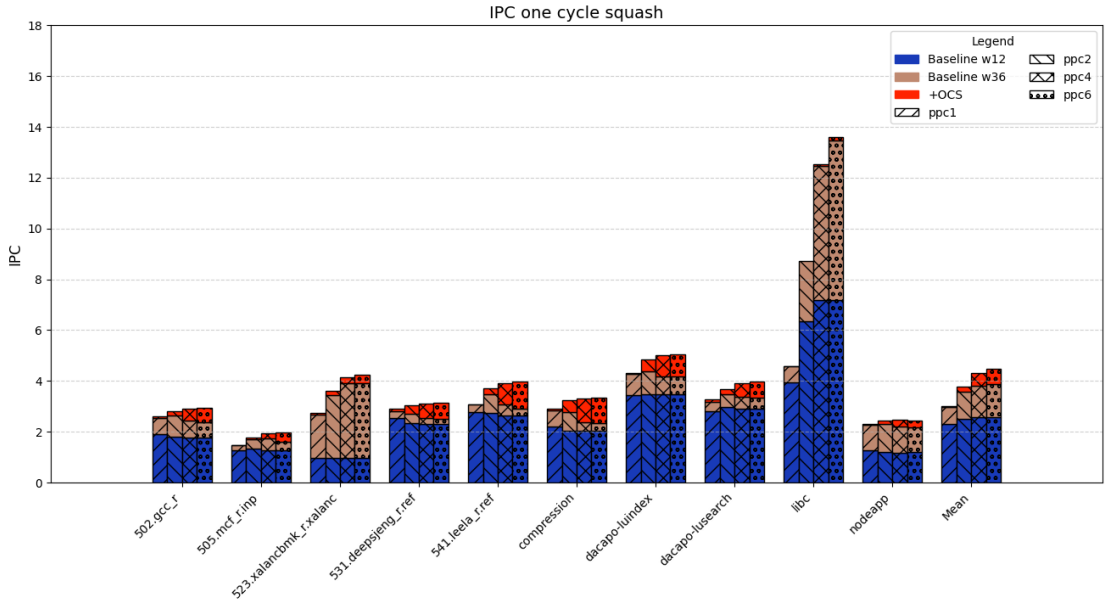


Figure 5.13: IPC: One cycle squash

5.3.2 Infinite TAGE to solve high MPKI

In order to reduce the number of squashes triggered by branch mispredictions, we must reduce the MPKI, this amounts to improving the accuracy of the BPU. To model such an accurate branch predictor, we use *infinite TAGE*. This configuration relies on a TAGE-SC-L [31] instance with tuned hyperparameters (as explained in Table 3.3) to allow the use of much more memory, enabling the branch predictor to use longer history tables. This increased storage budget should ultimately improve prediction accuracy. We take the previous configuration *one cycle squash* as a starting point and build *infinite TAGE* on top of it.

Figure 5.14 shows how the MPKI decreases using *infinite TAGE*. As a reminder, MPKI does not depend on the number of predictions per cycle, therefore, we no longer differentiate at this point. *Compression*'s MPKI drops close to 0, showing an impressive decrease of 99.8%. The average MPKI decrease is 38%. It is important to point out that increasing history table sizes, as we did with *infinite TAGE*, cannot improve the prediction accuracy of all branches. Hard-to-predict branches (for example, data-dependent branches) remain a challenge for the BPU, it is thus impossible to achieve perfect branch prediction (0 MPKI) for all benchmarks. Nevertheless, we still

expect an improvement in IPC.

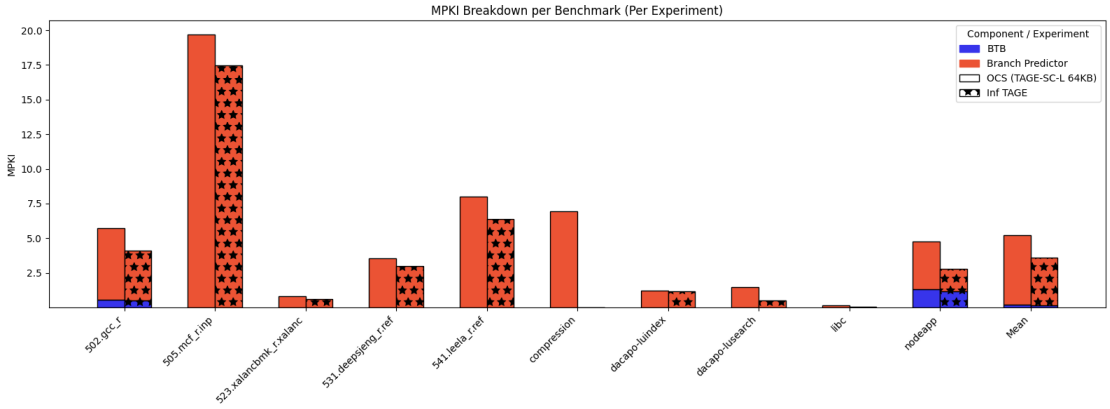


Figure 5.14: MPKI - TAGE-SC-L 64KB vs infinite TAGE

Figure 5.15 shows the IPC for *Infinite TAGE*. A general improvement over *one cycle squash* is noticeable. The average IPC is 3.24 for one prediction and 4.73 for six predictions. The speedup of multiple branch prediction does not change much compared to *one cycle squash*, with an average of 1.39X and a maximum value of 3.03X for *libc*.

The fact that *compression*'s IPC does not improve for more predictions per cycle despite its close to zero MPKI is explained by a bottleneck shift from branch mispredictions to backend bound, especially core bound and serializing stalls. We noticed an increase in serializing stalls bound from 2% to 14% and in core bound from 11% to 53% comparing the six predictions per cycle scenario without and with *infinite TAGE*, respectively. We explain this phenomenon by the fact that the previously high MPKI of *compression* was leading to the core spending most cycles squashing. Now that this bottleneck is removed with *infinite TAGE*, most of the instructions fetched are not squashed, however, this exposes the inherent bottlenecks of *compression*:

- It relies on serializing instructions that stall the rename stage and, in a chain effect, the previous stages, also forcing the ROB to drain before resuming instruction renaming, fundamentally limiting out-of-order execution.
- The instructions of *compression* are heavily dependent on each other, these dependency chains obviously limit the number of instructions that the core can execute out-of-order, which is reflected in the increasing core bound.

We consider *compression*'s case a good example of stacked bottlenecks: one bottleneck may hide another. Just as frontend bottlenecks (instruction fetch rate limitations) will

hide backend bottlenecks, the high MPKI of *compression* was hiding its inherent ILP limitations.

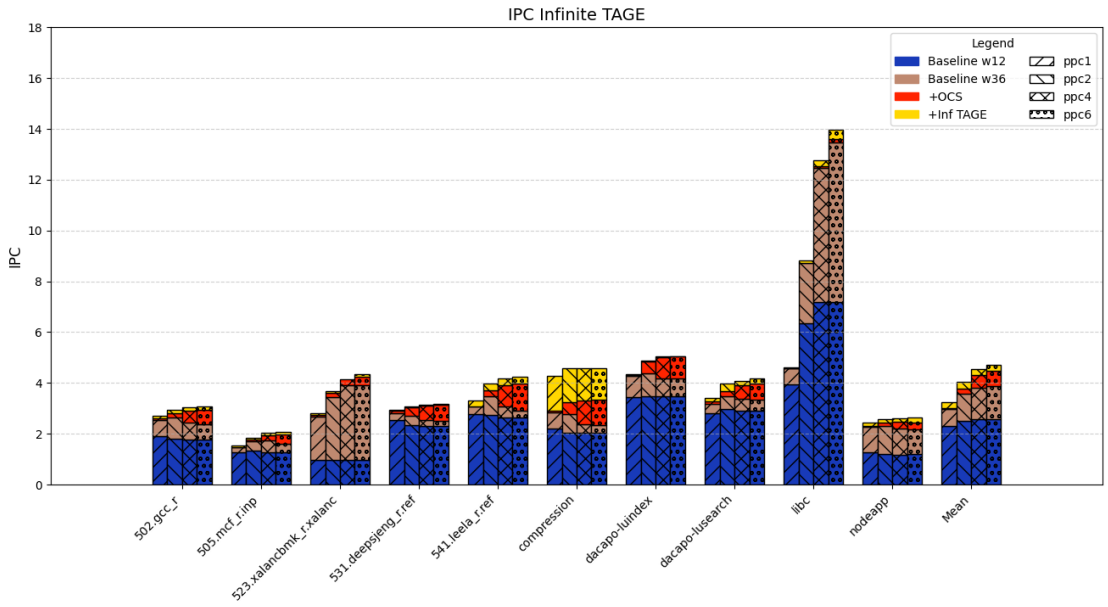


Figure 5.15: IPC: Infinite TAGE

We now move forward and focus on the effects of a better memory dependency prediction unit.

5.3.3 Infinite PHAST to reduce memory order violations

The effects of memory order violations are similar to those of branch mispredictions: the ROB needs to squash all the instructions from the wrong path. To reduce the squashes due to wrong memory dependency prediction, we replace the generic *store sets* memory dependency predictor of the O3 core with PHAST [18, 19], a state-of-the-art memory dependency predictor that has been evaluated on several workloads and is more accurate than *store sets*. However, as we aim to achieve the highest performance possible, we tuned its hyperparameters (as explained in Table 3.4) to improve its accuracy and get closer to a perfect memory dependency predictor. This configuration is referred to as *infinite PHAST*. We expect this to considerably reduce the number of memory order violations and ultimately improve the core's performance. As before, we added *infinite PHAST* to the previous configuration with *infinite TAGE*.

Figure 5.16 shows the number of memory order violations for *infinite PHAST* and the previous configuration that used *store sets*. The number of memory order violations

significantly decreased, demonstrating that PHAST is better at detecting memory dependencies. This result means that the total recovery penalty due to memory dependencies also decreases. Therefore, we expect this decrease in memory order violations to translate to some extent into IPC improvement.

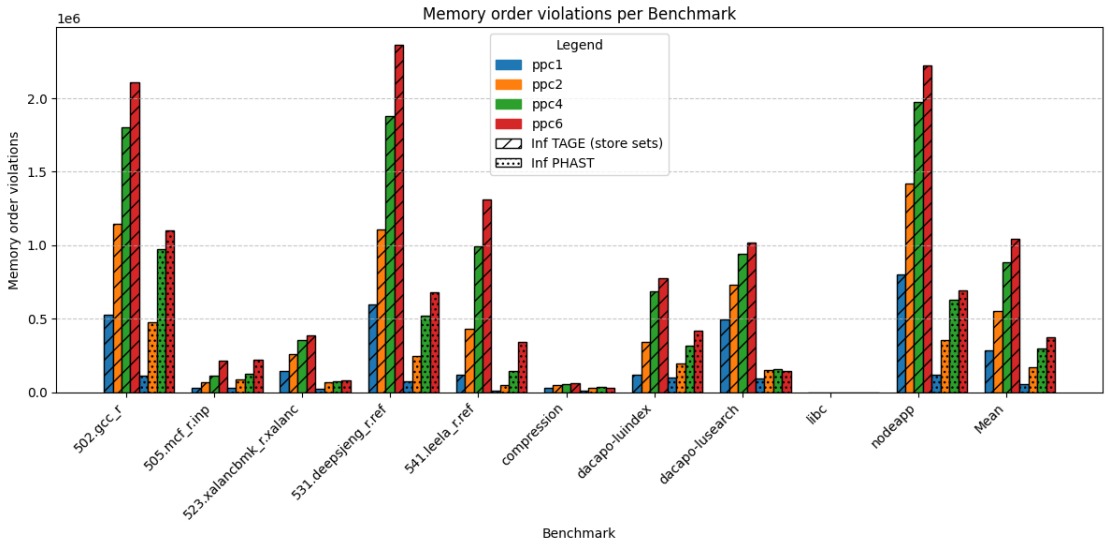


Figure 5.16: Memory order violations - Store sets vs infinite PHAST

As Figure 5.17 shows, most of the benchmarks benefit from this improved MDP accuracy in terms of IPC. A good example is *deepsjeng*, where the reduction in memory order violations strongly improves IPC. The average IPC is 3.53 for one prediction and 5.53 for six predictions. Looking at the speedup of six predictions per cycle over one prediction, it is clear that *libc* remains the benchmark with the most significant speedup, it however remains at 3.03X, since as Figure 5.16 shows, it did not have many memory order violations with *store sets* already. The average speedup is 1.52X, meaning a 9% increase over the 1.39X average speedup that *infinite TAGE* achieves.

Now that the bad speculation bottlenecks are all addressed, we end up with a configuration that enables performance improvement with multiple branch prediction across all benchmarks. This result not only shows the potential of multiple predictions per cycle but also highlights the relevance of speculative execution accuracy. Multiple branch prediction opens the door to speculative execution multiple blocks ahead. However, the prediction units (BPU, MDP) must be accurate enough so that speculative execution opportunities become actually valuable for the core.

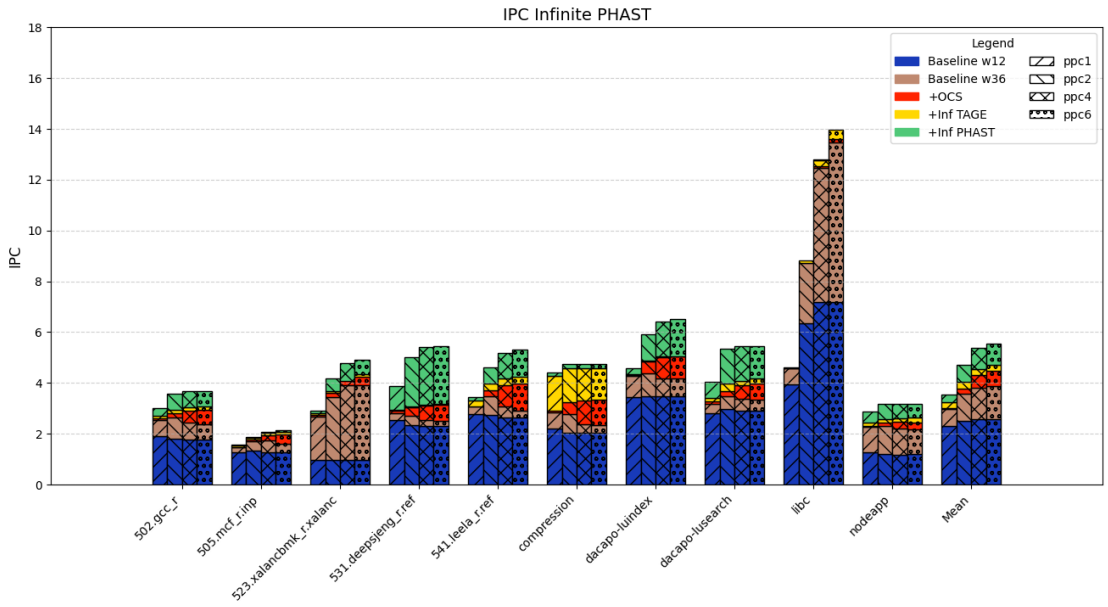


Figure 5.17: IPC: Infinite PHAST

Bad speculation is not the only bottleneck encountered by multiple branch prediction. Cache miss latency is also an important factor. Thus, the next step is to explore the performance opportunity of a low cache miss configuration.

5.3.4 Giant cache: lower data cache misses

We previously explained how multiple branch prediction leads to more simultaneous data cache misses and the possible effects on out-of-order execution. The obvious solution is to lower the number of cache misses. There are multiple approaches to achieve lower cache misses, ranging from data cache prefetchers to more complex cache hierarchies (for example, introduction of an L0 data cache). For our study, we chose simplicity of implementation and increased the size of the cache. We use a single level cache hierarchy, with an L1 data cache of 32MiB. The idea behind this unrealistically giant cache is to fit (almost) all the benchmark data in the L1 cache. This configuration aims to approximate a scenario without any cache misses. We refer to this scenario as *giant cache*, which is added on top of *infinite PHAST*, the previous configuration.

Figure 5.18 compares the L1 data cache misses between *giant cache* and *infinite PHAST*. Looking at this graph, it is safe to say that we are close to the goal of no cache misses for most of the benchmarks. We observe a 99% decrease for *xalancbmk* (this benchmark exhibits an increasingly high memory bound as Figure 5.5 shows), *libc*, and *mcf*, and on

average a 79.6% decrease in cache misses.

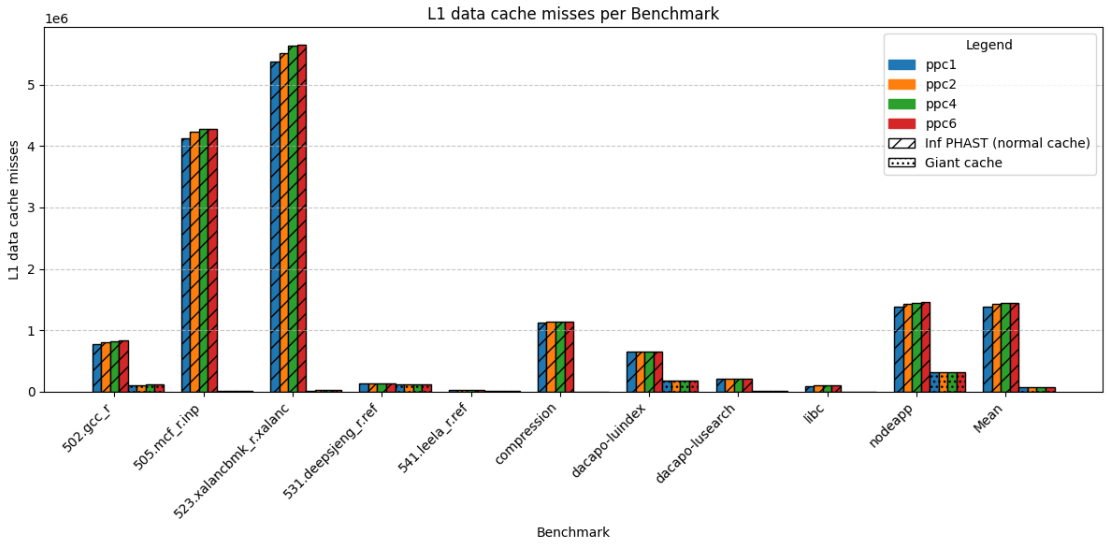


Figure 5.18: L1 Data cache misses: Normal cache vs Giant cache

This graph confirms that our *giant cache* configuration reduces data cache misses. The expectation is that such a configuration will be able to decrease the effect of data cache miss latency on multiple branch prediction performance.

Figure 5.19 shows the resulting IPC improvement of *giant cache* over *infinite PHAST*. There is an overall increase in IPC for most of the benchmarks, the average IPC is 3.87 for one prediction and 6.71 for six predictions, *xalancbmk* is, however, outstandingly better in this configuration. Its speedup for six predictions per cycle over one prediction is 3.02X, not far from the 3.04X achieved by *libc*. The average speedup is 1.71X, representing a 12.5% increase over *infinite PHAST* and more interestingly a 42.5% increase over the baseline *w36*.

The *giant cache* configuration combines all the optimizations we tried to improve multiple branch prediction performance, assuming a fixed instruction window size. The 42.5% increase in the average speedup of this configuration over the baseline brings us to the following conclusion: branch prediction, memory dependency prediction, and cache misses are bottlenecks already present in a generic out-of-order CPU architecture. However, these bottlenecks become even more important when trying to leverage more ILP with more speculative execution using multiple branch prediction. If these bottlenecks are not addressed correctly, a lot of performance improvement opportunities are wasted.

As we mentioned earlier in this chapter, the first limitation of the O3 core's backend

(and of out-of-order CPUs in general) is its ability to rename instructions, which might be constrained by multiple factors like register availability, instruction window size, rename width, and other key structures. The final step of our performance evaluation approach is therefore to remove this bottleneck, this is what we address in the next part.

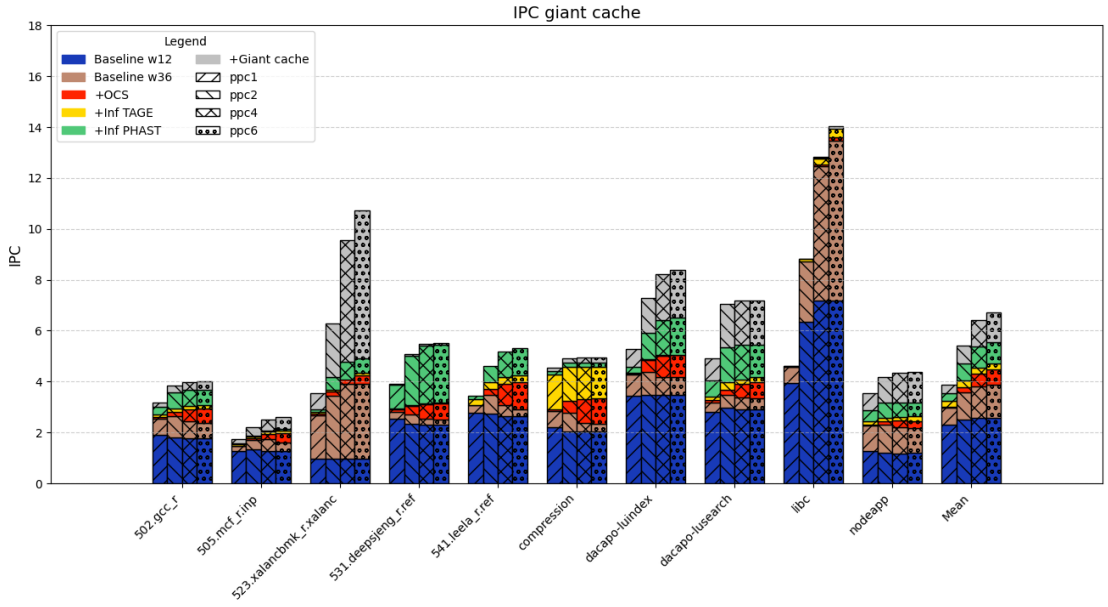


Figure 5.19: IPC: Giant cache

5.3.5 Scaling up the already optimized setup

The O3 core's rename width defines how many instructions can be renamed in one cycle. It is an upper limit. For an instruction to be renamed, enough registers and free space in the ROB and in the IQ must be available. More specifically, there also needs to be slots free in the load store queue (LSQ) for load and store instructions. We need to scale all these structures and buffers to completely remove the instruction renaming bottleneck. The number of functional units also needs to be scaled to execute more instructions, otherwise, it would be a bottleneck. This comprehensive scaling is precisely what we do in the configuration *bigger inst window*, adding all the above-mentioned modifications on top of *giant cache*. Allowing more instructions to rename should theoretically fill the ROB faster than before, allowing more speculative execution. We also scale the frontend capabilities to a similar extent. This scaling is necessary to keep the core configuration consistent and should also improve the instruction fetch rate, giving more out-of-order

execution opportunities to the backend. The exact configuration details are in Table 3.2.

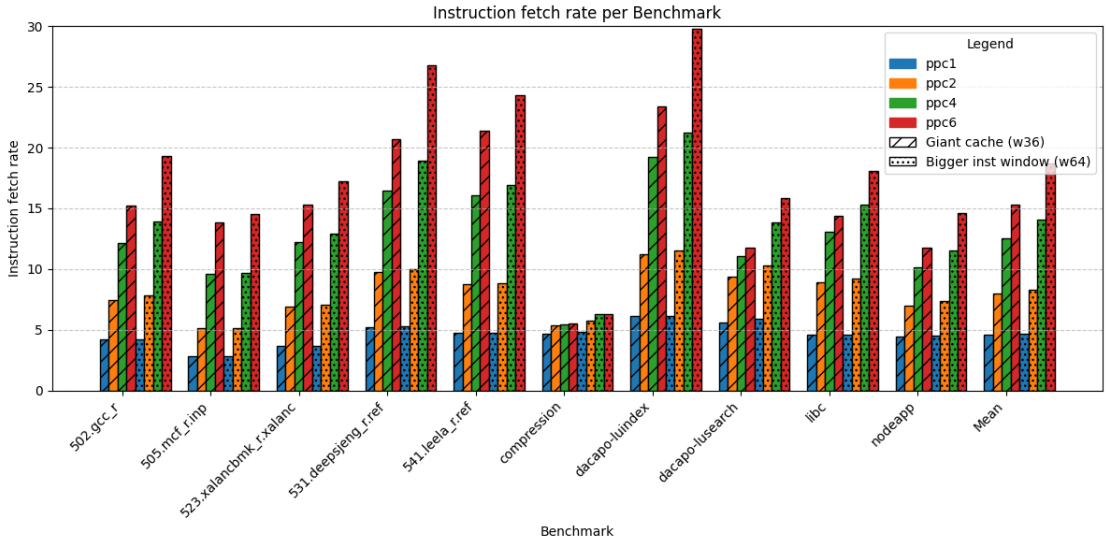


Figure 5.20: Instruction fetch rate: Bigger inst window

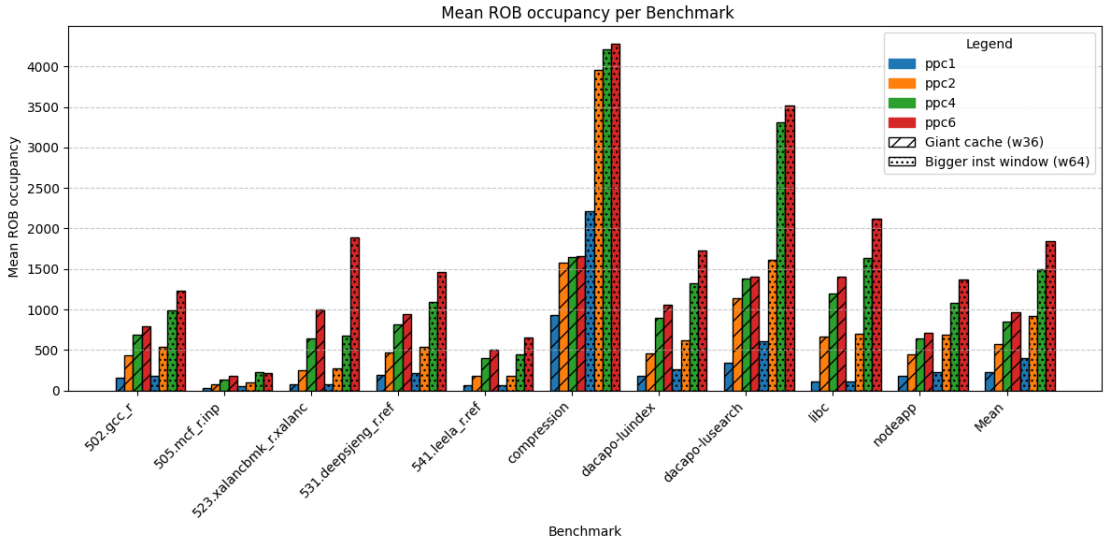


Figure 5.21: ROB occupancy: Bigger inst window

Figures 5.20 and 5.21 both confirm our hypothesis on the effect of this new configuration on the instruction fetch rate as well as on the ROB occupancy. Instruction fetch

rate improves for most of the benchmarks.

It is interesting to notice that there is almost no improvement in instruction fetch rate for *compression*. This stagnation is due to the high number of serializing stalls, stalling the rename stage and subsequently the decode and fetch stages as well. The increase in ROB occupancy means that the backend has more room to exploit the ILP of the benchmarks. This increase should translate to IPC improvement.

Figure 5.22 shows the IPC improvement of *bigger inst window* over *giant cache*. The IPC improvement of *libc* for six predictions per cycle stands out, whereas the other benchmarks show only modest improvement. The average IPC for six predictions is 7.15, and 3.89 for one prediction. The speedup for six predictions per cycle over one prediction for *libc* is 3.72X, and the average is 1.79X. This represents only a 4.6% improvement over *giant cache*, which is low considering that *bigger inst window* has significantly more resources.

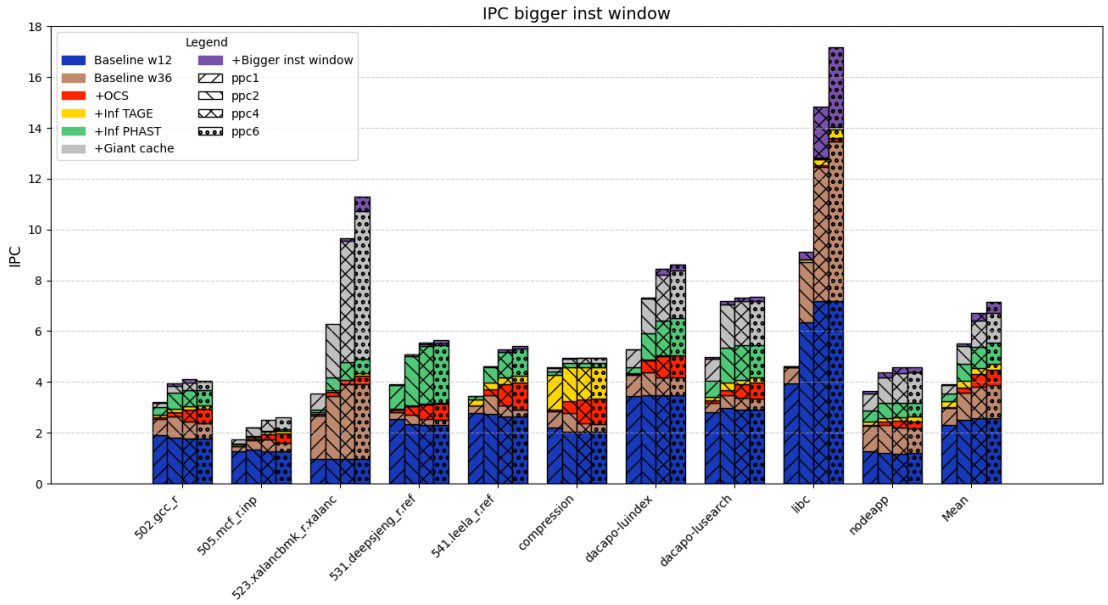


Figure 5.22: IPC: Bigger inst window

Figure 5.23 gives an overview of each benchmark's main limiting factors. The frontend bound might increase for some benchmarks as the pipeline is wider (this means more fetch slots available), but we do not make more than six predictions per cycle. This increase in frontend bound is not harmful to the backend performance, and it is necessary to keep the same multiple branch prediction scenarios to compare the results of *bigger inst window* with the previous results.

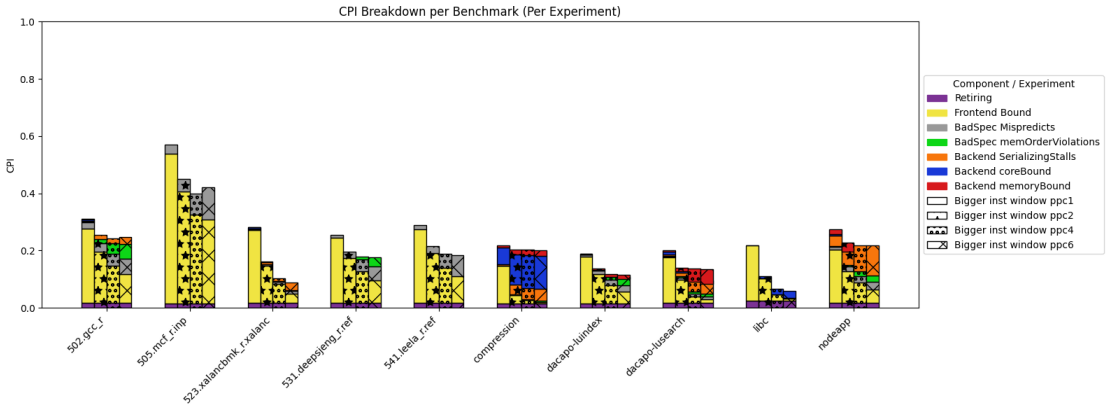


Figure 5.23: L2 CPI Stacks: Bigger inst window

It is more interesting to look at the difference between *libc* and the benchmarks exhibiting low improvement: they have high shares of bad speculation, serializing stalls, or memory bound.

We explain the performance improvement of *libc* by its inherent characteristics: low MPKI and a predictable memory access pattern (few memory order violations and data cache misses). However, for other benchmarks exhibiting bottlenecks like branch mispredictions (for example, *mcf* with around 17 MPKI), memory order violations, and serializing stalls, the improvement is rather limited.

The low overall performance improvement of *bigger inst window* over *giant cache* confirms our explanation and demonstrates that increasing the instruction window and scaling the surrounding backend infrastructure does not make sense if the speculative execution is still limited by one or more of the above-mentioned bottlenecks.

5.3.6 Reducing penatly or better predictors?

Previous sections showed that despite our attempts to improve branch and memory dependency prediction, it is hard to achieve perfect predictors. For this reason, we want to highlight the importance of reducing the recovery penalty as it can mitigate the effects of mispredictions, as we did in *one cycle squash*.

To do so, we run the *infinite TAGE only* configuration, using the same branch predictor as *infinite TAGE*, but added on top of the baseline *w36*, so with a *normal squash width*. For instance, 36 in the *w36* configuration. We directly compare it to *one cycle squash*.

As Figure 5.24 shows, *one cycle squash* is better than *infinite TAGE only* for multiple branch prediction for eight of our ten benchmarks, with a less good branch predictor. Another observation is that *infinite TAGE only* does not solve the problem of diminishing

performance with more predictions per cycle despite the 39% decrease in MPKI. The better performance of *infinite TAGE only* for *compression* and *libc* is solely explained by the fact that in both cases the MPKI is close to zero due to the better branch predictor. The takeaway is that since perfect branch prediction is hard to achieve, reducing the misprediction penalty is the better solution to improve performance even with a high MPKI *independently* of the workload.

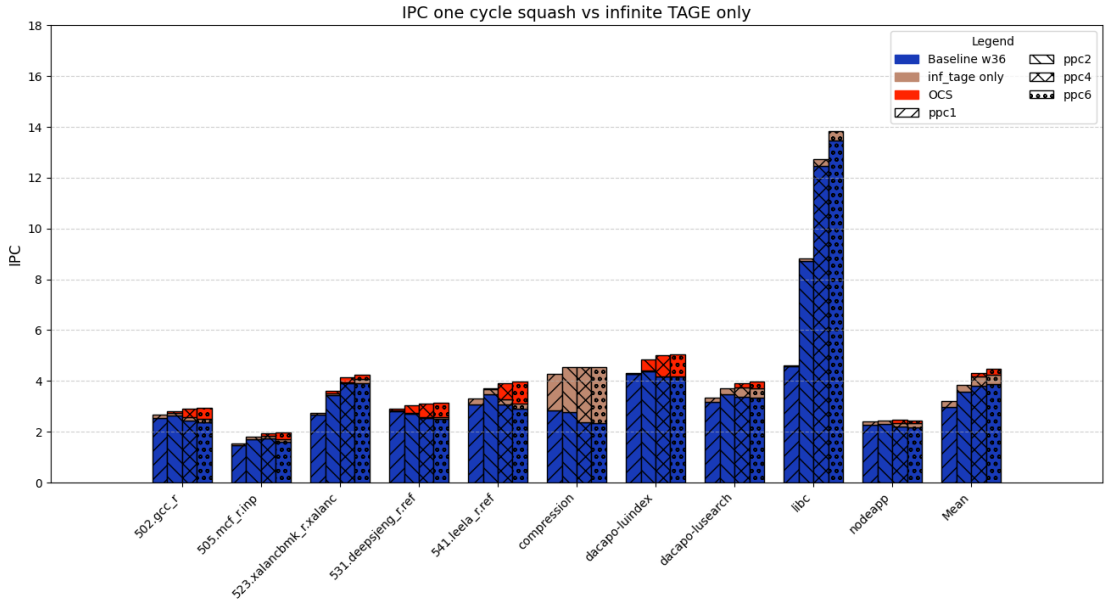


Figure 5.24: IPC: Infinite TAGE only vs one cycle squash

5.4 Limitations

One of the goals of this chapter was to determine the theoretical performance upper limit of the O3 core - the maximum IPC reachable. We analyzed the backend behavior and identified branch prediction and memory dependency prediction as critical bottlenecks limiting the performance improvements from multiple branch prediction's increased speculative execution. We then successively attempted to remove these bottlenecks.

Unfortunately, it was impossible to completely eliminate these bottlenecks, because perfect predictors are not available in gem5. Therefore, one of the main limitations of our results compared to other ILP research works is our inability to model perfect branch and memory dependency prediction. It could significantly impact performance

and provide a more accurate assessment of the theoretical limits of multiple branch prediction potential.

Additionally, gem5's ISA models are inaccurate. While serializing instructions exist in real hardware, some serializing stalls in gem5 result from incomplete renaming implementations rather than actual architectural requirements. This mismatch means that, depending on the workload, the O3 core might exhibit significantly more serializing stalls than real hardware. Since serializing stalls significantly constrain out-of-order execution, this leads to lower performance than in real processors. We did not evaluate the magnitude of this difference between gem5 and real hardware, but we consider this a significant limitation of both gem5 and our results. Due to time constraints, we could not identify and fix all the affected instructions.

Other limitations include our decision not to investigate BTB organizations, which might positively impact branch prediction performance. Also, we did not explore instruction scheduler optimizations, which may improve the resulting IPC. Finally, the delay between the stages was one cycle across all our experiments. As we conduct an opportunity study, this is an advantage because the stages are faster. However, this could accentuate the observed misprediction recovery penalty. More delay between the stages would hide some of the latency. The penalty could be lower on a configuration more similar to real hardware with realistic delays between stages.

6 Related work

The two main categories of work related to this thesis are multiple branch prediction and instruction-level parallelism. In this Chapter, we present two of the most relevant papers in each category and compare them to the results we achieved in this thesis.

6.1 Multiple branch prediction

Yeh et al. [40] presented one of the foundational works in multiple basic block fetching with their mechanism for predicting multiple branches and fetching multiple non-consecutive basic blocks each cycle. Their approach introduced three key components: a Multiple Branch Two-level Adaptive Branch Predictor able to predict multiple branches per cycle, a branch address cache (nowadays BTB) to provide addresses of basic blocks following predicted branches, and a high-bandwidth instruction cache configuration. The branch address cache stores up to 6 fetch addresses for two-branch prediction (2 primary and 4 secondary basic block addresses) or 14 addresses for three-branch prediction. Their results showed significant improvements in instruction fetch rate, with integer benchmarks improving from 3.0 to 4.2 and 4.9 instructions per cycle when fetching 1, 2, and 3 basic blocks, respectively, while floating-point benchmarks improved from 6.6 to 7.1 and 8.9 instructions per cycle.

Seznec et al. [32] introduced the concept of two-block ahead branch prediction, which differs fundamentally from previous approaches by using information from the current instruction block to predict the block following the next instruction block, rather than just the immediate next block. This approach offers advantages for both "brainiac" processors (wide-issue, parallel execution) and "speed demon" processors (high clock rate). For brainiac processors, it enables efficient prediction of two instruction blocks in a single cycle by using both current fetch addresses to predict two subsequent blocks. For speed demon processors, it allows pipelining of the branch prediction process to achieve higher clock rates or accommodate larger prediction structures without introducing pipeline bubbles. Their experimental results demonstrated that the two-block ahead branch predictor achieves equivalent prediction accuracy to conventional one-block-ahead predictors (within 0.30% misprediction rate difference across SPEC92 benchmarks) while providing significant performance improvements for wide-dispatch

processors. For 8-wide processors, double I-fetch mechanisms provided 20-40% performance improvement over single I-fetch processors. The two-block ahead BTB requires only slightly higher associativity (4-way vs 2-way) compared to conventional BTBs while maintaining similar storage requirements.

In comparison, our work differs from these practical multiple branch prediction designs by focusing on evaluating the theoretical opportunity and limits of multiple branch prediction to improve instruction fetch rate and address the single prediction bottleneck as core pipeline width increases. Our study provides an analysis of the fetch rate improvements achievable with multiple predictions, ranging from single to six predictions per cycle. Additionally, we identified and characterized the fetch buffer bottleneck in gem5's O3 core when implementing multiple branch prediction mechanisms an proposed a design to address it. Our results demonstrate substantial improvements in instruction fetch rate, achieving speedups from 4.42 to 15.18 (3.4X) instructions per cycle on average with six predictions per cycle in the *w36* configuration, and from 3.48 to 6.11 (1.74X) for the *w12* configuration. These results corroborate the findings of prior work while providing new insights into the scalability and practical limitations of multiple branch prediction in modern processor simulators.

6.2 Instruction-level parallelism

Wall [35] established the theoretical limits of ILP through trace-driven simulation. His "Perfect" model with ideal branch prediction, infinite register renaming, and perfect alias analysis (memory dependency prediction), parallelism could reach up to 60 instructions per cycle for individual programs, with an average of around 25. However, even these impossibly good techniques rarely exceeded an average parallelism of 7 across the benchmark suite. His ambitious software-style model with static prediction and 2K instruction windows achieved average parallelism closer to 9, while realistic implementations achieved roughly half the theoretical limits.

Chadwick et al. [10] updated this analysis for modern processors using a dynamic dependency graph (DDG) methodology that measures the longest path through program dependencies, incorporating modern prediction techniques (branch, memory-dependence, and value prediction) that can break dependency edges when successful. Using SPEC CPU 2017 benchmarks, they found current realistic ILP around 4.8 with projections that 2030s processors could achieve 2.2× improvement, meaning around 10.6. Furthermore, their analysis revealed that with infinite instruction window and pipeline width, an average of 40.3 was achievable. However, A third of workloads receive most of their performance scaling from the 2030 projection. They suggested the need for new approaches beyond traditional ILP exploitation.

Compared to these previous studies, the main difference with our work is that we use gem5. This execution-based simulator models not only branch prediction and register renaming but the entire pipeline behavior, from instruction fetching until retirement. Gem5 models factors that degrade IPC, such as cache misses, pipeline stalls, branch misprediction penalties, etc.

Our study estimates ILP under conditions more similar to real hardware. Additionally, it evaluates whether multiple branch prediction can help real processors achieve this ILP as pipelines grow wider and cores need to fetch more instructions to fill larger instruction windows. The advantage of execution-based simulation is that results are comparable to real processors; however, this typically yields lower IPC than trace-based simulators or DDGs estimate. Our *bigger inst window* configuration, with an instruction window similar to the future processors described by Chadwick et al. [10], achieves an average IPC of 7.14, which is reasonably close to their results considering the additional IPC-limiting factors that gem5 models.

7 Conclusion

As pipelines grow wider and instruction windows grow larger over the years [8, 20], advanced instruction fetching mechanisms become crucial. These mechanisms must efficiently utilize pipeline width and fill instruction windows faster. In this thesis, we highlighted the existing bottlenecks of the O3 core, implemented multiple branch prediction in gem5, and proposed an FTQ design that enables better performance. We then evaluated the potential of multiple branch prediction for the core's frontend. With the *w12* configuration, similar to current processors, our approach achieved an average of 6.11 instructions fetched per cycle with six predictions compared to 3.48 for one prediction. Moreover, on our *w36* configuration, which corresponds to possible future processors, we reached an average of 15.2 instructions fetched per cycle with six predictions compared to 4.41 for one prediction, achieving a substantial speedup of 3.4X. Our evaluation demonstrates the potential of multiple branch prediction, making it a suitable solution to ensure a steady instruction supply, especially for wide-pipeline architectures, which we expect will become standard in the future.

We then evaluated the impact of this increased instruction throughput on the core's backend. While we observed diminished performance for multiple branch prediction at first, we were able to identify the critical components and parameters leading to these results. Notably, the ROB squash penalty, predictor accuracy, and cache miss latency were the main bottlenecks. While these are also bottlenecks in single prediction architectures, we explained why they become even more critical in the context of multiple branch prediction. We then attempted to solve the bottlenecks step-by-step, building a backend that could leverage the more speculative execution opportunities enabled by multiple branch prediction and the large instruction window. We highlighted the squash width as an important parameter for the O3 core to reduce the ROB's squash penalty.

With our best configuration (*bigger inst window*), featuring a large instruction window, predictors with infinite storage budget, and a memory hierarchy that minimizes cache misses, we were able to achieve an average IPC of 7.15 with six predictions per cycle compared to 3.89 with one prediction. We believe these results are promising considering the limitations of gem5 we encountered. Overall, we showed that multiple branch prediction can significantly increase IPC, however, it can also increase misprediction recovery. Therefore, it requires a sufficiently robust architecture to reach its full

potential. Additionally, we showed that reducing the misprediction recovery penalty is the crucial first step, yielding better results than a branch predictor with an infinite storage budget (*infinite TAGE only*). The full implementation in gem5 is available here: `https://github.com/dhschall/gem5-fdp/tree/pf-rework-multi-pred`

In this thesis, we demonstrated the potential of multiple branch prediction to improve instruction fetch rate, especially on scaled pipelines. However, we did not implement a realistic multiple branch prediction scheme, as our focus was to estimate the performance opportunity. Therefore, future research should focus on building a realistic, state-of-the-art multiple branch prediction design applicable to real hardware. Such a design could potentially build upon the FTQ design we proposed.

Additionally, we studied the effect of increased instruction throughput on the backend and highlighted major bottlenecks. We found that reducing the recovery penalty was the most important factor for our workloads to improve speculative execution performance. Future work could investigate practical ways to reduce the penalty in real hardware, such as dynamic instruction reuse or a squash latency hiding mechanism.

Moreover, to address the limitations of our work, implementing perfect predictors (branch, memory dependency, value) in gem5 could significantly improve the results of our ILP study and provide a stronger foundation for future research in this area. Additional improvements to gem5's O3 core could also enhance simulation accuracy. These include implementing more advanced instruction scheduling techniques and configuring the ISA models to match real hardware implementations more closely. Such enhancements would make the simulator more representative of actual processors and improve the overall accuracy of performance evaluations.

Abbreviations

FDP fetch-directed instruction prefetching

BAC Branch and address calculation

BTB branch target buffer

BPU branch prediction unit

FT fetch target

FTQ fetch target queue

ROB reorder buffer

IQ instruction queue

MDP memory dependency predictor

ILP instruction-level parallelism

MPKI mispredictions per kilo instructions

RAS return address stack

PC program counter

IPC instructions per cycle

CPI cycles per instruction

LSQ load store queue

List of Figures

List of Tables

Bibliography

[1] N. Adiga, J. Bonanno, A. Collura, M. Heizmann, B. R. Prasky, and A. Saporito. "The IBM z15 High Frequency Mainframe Branch Predictor Industrial Product." In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 2020, pp. 27–39. DOI: 10.1109/ISCA45697.2020.00014.

[2] A. Akram and L. Sawalha. "Validation of the gem5 Simulator for x86 Architectures." In: *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 2019, pp. 53–58. DOI: 10.1109/PMBS49563.2019.00012.

[3] K. Asanović, D. A. Patterson, and C. Celio. "The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor." In: 2015.

[4] S. Bambou and D. Schall. *Gem5 configuration file for server workloads*. URL: https://github.com/dhschall/gem5-svr-bench/blob/multi-pred/gem5-configs/fs-fdp-multi.py.

[5] S. Bambou and D. Schall. *Gem5 configuration file for SPEC workloads*. URL: https://github.com/dhschall/gem5-svr-bench/blob/multi-pred/gem5-configs/spec-run.py.

[6] S. Bambou and D. Schall. *Gem5 O3 core configuration file*. URL: https://github.com/dhschall/gem5-svr-bench/blob/multi-pred/gem5-configs/util/simulation_utils.py.

[7] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. "The DaCapo benchmarks: java benchmarking development and analysis." In: *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. OOPSLA '06. Portland, Oregon, USA: Association for Computing Machinery, 2006, pp. 169–190. ISBN: 1595933484. DOI: 10.1145/1167473.1167488.

[8] D. Burger. "Designing Ultra-large Instruction Issue Windows." In: *Advances in Computer Systems Architecture*. Ed. by A. Omondi and S. Sedukhin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 14–20. ISBN: 978-3-540-39864-6.

[9] C. Carvalho. "The Gap between Processor and Memory Speeds." In: 2002.

[10] A. Chadwick, M. Erdos, U. Bora, A. Bhosale, B. Lytton, Y. Guo, R. Cooper, G. Gabrielli, and T. Jones. "The Future of Instruction-Level Parallelism (ILP)." In: May 2025, pp. 350–352. DOI: 10.1109/ISPASS64960.2025.00040.

[11] B. Cohen, M. Subramony, and M. Clark. *AMD Next Generation "Zen 5" Core*. URL: https://hc2024.hotchips.org/assets/program/conference/day2/24_HC2024.AMD.Cohen.Subramony.final.pdf.

[12] G. Cozma and Camacho. *Zen 5's 2-Ahead Branch Predictor Unit: How a 30 Year Old Idea Allows for New Tricks*. URL: https://chipsandcheese.com/p/zen-5s-2-ahead-branch-predictor-unit-how-30-year-old-idea-allows-for-new-tricks.

[13] W. Damm and A. Pnueli. "Verifying Out-of-Order Executions." In: *Advances in Hardware Design and Verification: IFIP TC10 WG10.5 International Conference on Correct Hardware and Verification Methods, 16–18 October 1997, Montreal, Canada*. Ed. by H. F. Li and D. K. Probst. Boston, MA: Springer US, 1997, pp. 23–47. ISBN: 978-0-387-35190-2. DOI: 10.1007/978-0-387-35190-2_3.

[14] S. Eyerman, L. Eeckhout, and K. De Bosschere. "Efficient Design Space Exploration of High Performance Embedded Out-of-Order Processors." In: *Proceedings of the Design Automation & Test in Europe Conference*. Vol. 1. 2006, pp. 1–6. DOI: 10.1109/DATE.2006.243735.

[15] Google. *Benchmarking suite for Google workloads*. URL: https://github.com/google/fleetbench.

[16] Y. Ishii, J. Lee, K. Nathella, and D. Sunwoo. "Rebasing Instruction Prefetching: An Industry Perspective." In: *IEEE Computer Architecture Letters* 19.2 (2020), pp. 147–150. DOI: 10.1109/LCA.2020.3035068.

[17] R. Kessler, E. McLellan, and D. Webb. "The Alpha 21264 microprocessor architecture." In: *Proceedings International Conference on Computer Design. VLSI in Computers and Processors (Cat. No.98CB36273)*. 1998, pp. 90–95. DOI: 10.1109/ICCD.1998.727028.

[18] S. S. Kim and A. Ros. "Effective Context-Sensitive Memory Dependence Prediction." In: *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 2024, pp. 515–527. DOI: 10.1109/HPCA57654.2024.00045.

[19] S. S. Kim and A. Ros. *Phast-Gem5 repository*. URL: https://gitlab.com/muke101/gem5-phast.

[20] F. Latorre, G. Magklis, J. González, P. Chaparro, and A. González. "CROB: Implementing a Large Instruction Window through Compression." In: *Transactions on High-Performance Embedded Architectures and Compilers III*. Ed. by P. Stenström. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 115–134. ISBN: 978-3-642-19448-1. DOI: 10.1007/978-3-642-19448-1_7.

[21] M. Mannino. *Simpoints in Gem5*. URL: https://mircomannino.github.io/posts/simulation/1_gem5_simpoints.html.

[22] A. Mohammad, U. Darbaz, G. Dozsa, S. Diestelhorst, D. Kim, and N. S. Kim. "dist-gem5: Distributed simulation of computer clusters." In: *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2017, pp. 153–162. DOI: 10.1109/ISPASS.2017.7975287.

[23] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt. "Runahead execution: an alternative to very large instruction windows for out-of-order processors." In: *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.* 2003, pp. 129–140. DOI: 10.1109/HPCA.2003.1183532.

[24] A. Pellegrini, N. Stephens, M. Bruce, Y. Ishii, J. Pusdesris, A. Raja, C. Abernathy, J. Koppanalil, T. Ringe, A. Tummala, J. Jalal, M. Werkheiser, and A. Kona. "The Arm Neoverse N1 Platform: Building Blocks for the Next-Gen Cloud-to-Edge Infrastructure SoC." In: *IEEE Micro* 40.2 (2020), pp. 53–62. DOI: 10.1109/MM.2020.2972222.

[25] 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. DOI: 10.1109/MICRO.1999.809439.

[26] G. Reinman, B. Calder, and T. Austin. "Optimizations enabled by a decoupled front-end architecture." In: *IEEE Transactions on Computers* 50.4 (2001), pp. 338–355. DOI: 10.1109/12.919279.

[27] D. Schall. *Development repository for Fetch Directed Instruction Prefetching (FDP) in gem5*. URL: https://github.com/dhschall/gem5-fdp.

[28] D. Schall. *gem5 Server Benchmarks*. URL: https://github.com/dhschall/gem5-svr-bench/tree/multi-pred.

[29] D. Schall, A. Sandberg, and B. Grot. "The Last-Level Branch Predictor." In: *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2024, pp. 464–479. DOI: 10.1109/MICRO61859.2024.00042.

[30] A. Seznec. "A 64-Kbytes ITTAGE indirect branch predictor." In: (June 2011).

[31] A. Seznec. "TAGE-SC-L Branch Predictors Again." In: *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*. Seoul, South Korea, June 2016.

[32] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud. "Multiple-block ahead branch predictors." In: *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS VII. Cambridge, Massachusetts, USA: Association for Computing Machinery, 1996, pp. 116–127. ISBN: 0897917677. DOI: 10.1145/237090.237169.

[33] A. Seznec and P. Michaud. "A case for (partially) TAgged GEometric history length branch prediction." In: *Journal of Instruction-level Parallelism - JILP* 8 (Feb. 2006).

[34] SPEC. *SPECrate 2017 Integer*. URL: https://www.spec.org/cpu2017/Docs/index.html#intrate.

[35] D. W. Wall. "Limits of instruction-level parallelism." In: *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS IV. Santa Clara, California, USA: Association for Computing Machinery, 1991, pp. 176–188. ISBN: 0897913809. DOI: 10.1145/106972.106991.

[36] S. Wallace and N. Bagherzadeh. "Multiple branch and block prediction." In: *Proceedings Third International Symposium on High-Performance Computer Architecture*. 1997, pp. 94–103. DOI: 10.1109/HPCA.1997.569645.

[37] Wikipedia. *Intel's Golden Cove*. URL: https://en.wikipedia.org/wiki/Golden_Cove.

[38] O. Yaşar, D. Schall, and P. Bhatotia. *"A Top-Down Methodology for gem5" at gem5 Workshop at ISCA 2025*. URL: https://www.gem5.org/events/isca-2025.

[39] A. Yasin. "A Top-Down method for performance analysis and counters architecture." In: *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2014, pp. 35–44. DOI: 10.1109/ISPASS.2014.6844459.

[40] T.-Y. Yeh, D. T. Marr, and Y. N. Patt. "Increasing the instruction fetch rate via multiple branch prediction and a branch address cache." In: *Proceedings of the 7th International Conference on Supercomputing*. ICS '93. Tokyo, Japan: Association for Computing Machinery, 1993, pp. 67–76. ISBN: 089791600X. DOI: 10.1145/165939.165956.

[41] J. Zhao and A. Gonzalez. "Sonic BOOM: The 3rd Generation Berkeley Out-of-Order Machine." In: 2020.