SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Mitigating Branch Predictor Latency with Hierarchical Design — an Analysis

Phillip Assmann

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Mitigating Branch Predictor Latency with Hierarchical Design — an Analysis

Verringerung der Latenz von Sprungvorhersagen durch hierarchisches Design — eine Analyse

Author:	Phillip Assmann
Examiner:	Prof. Dr. Pramod Bhatotia
Supervisor:	Dr. David Schall
Submission Date:	15.07.2025

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

Munich, 15.07.2025 Phillip Assmann

Acknowledgments

I want to thank my advisor, Dr. David Schall, for his support and guidance throughout the development of this thesis. His expertise on the topic and constant feedback were very valuable and made this work possible.

I would also like to express my gratitude to Prof. Dr. Bhatotia for the opportunity to explore this topic at the chair.

Special thanks to my family for listening to my numerous and sometimes incomprehensible remarks about the details of branch prediction; you helped me push this thesis to completion.

Abstract

In modern processors, the branch predictor is a crucial component, which enables the effective use of large pipelines and wide instruction windows, anticipating program control flow and keeping the frontend on the correct path. Even after decades of research, the ever-increasing instruction footprint of large server workloads still poses a problem to modern branch predictors, overwhelming their storage-constrained history tables and the branch target buffer (BTB). While it is generally assumed that increasing the size of the branch predictor tables can mitigate this bottleneck, the latency and power constraints of these components make it challenging to scale up without nullifying the obtained accuracy gains due to late predictions. To improve the overall latency behavior of modern branch predictors, chip designers often use an overriding design with multiple predictors, where the result of the fast and small primary predictor is corrected ("overridden") by a more accurate estimation later. While this approach can hide significant amounts of latency, prior work suggests that it may also reach a scaling limit at some point, as predictor accuracy does not grow strictly proportional to storage capacity. A recent design, "The Last-Level Branch Predictor" (LLBP) [18], aims to overcome this scaling ceiling, improving the accuracy of a moderately-sized predictor by prefetching additional patterns from a large backing storage. Because the global history patterns used by modern predictors are constructed via multiple hash functions, they are scattered in storage and exhibit low spatial locality, making block prefetching or paging difficult. Instead, LLBP groups the patterns by execution context, formed by hashing the preceding call chain of unconditional branches into a fingerprint, allowing for precise prefetching along the execution flow. In this thesis, we provide an insight into the impact of predictor latency on overall system performance, and compare the approaches of overriding prediction and LLBP, to answer the question whether they can close the performance gap between branch predictor latency and accuracy.

Contents

1 Introduction

As Moore's law slows down, the focus of processor research and industry has shifted to improvements in microarchitecture to achieve higher performance in each new CPU generation, through wide and deep pipelines and large instruction windows, maximizing the benefits of superscalar execution.

One of the most critical components for instruction-level parallelism across large pipelines is the branch predictor. It keeps the frontend of the processor on the correct execution path, allowing the continuous fetching of future instructions in the presence of uncertainty caused by branch instructions, which can redirect the control flow of the program.

While state-of-the-art branch predictors like TAGE-SC-L [19] already achieve a high level of accuracy in many programs, they still experience a considerable amount of mispredictions in large server workloads, where the available storage capacity is too small to track all branches present in the large codebase. Consequently, recent research has found that a state-of-the-art Intel Sapphire Rapids server CPU, which contains a sophisticated branch predictor, still wastes over 9% of cycles due to mispredictions when running modern server workloads [18]. This number could grow significantly in the future, with Intel estimating that up to 50% of performance opportunity could be locked behind advances in branch prediction when scaling the pipeline of their current Skylake processor by 4x [11].

Even though proposals for architectural changes to the branch predictor have shown promising results [15] [5], it is generally assumed that a majority of mispredictions in large server workloads experienced by state-of-the-art branch predictors could be mitigated by providing additional storage capacity to the predictor [10].

The main issue with scaling the branch predictor is, however, that it lies on the critical path of the processor frontend, which is sensitive to latency. As with any type of storage, increasing the predictor storage budget leads to a higher access latency and energy cost to retrieve the necessary history information, forcing a painful compromise between accuracy and latency.

This tradeoff is well known from other areas of computer architecture, most notably in the memory hierarchy, where the disparity between core and memory frequencies has led to the creation of the hierarchical caching structure as we know it today. It allows the use of large amounts of memory without unacceptable access latency by caching spatially related blocks in smaller, faster memory components before they are needed.

However, applying the same hierarchical structure to the branch predictor has proven difficult, as the history patterns, which are needed for prediction, have very low spatial locality and cannot be prefetched effectively with traditional techniques, which degrades the performance of common hierarchical approaches like paging [16] [18].

Instead, many modern processors use multiple branch predictors, with increasing storage size and associated latency. The pipeline fetches instructions based on an initial guess of the fastest predictor, which can then be "overridden" later by a slower, more accurate predictor, reducing the stalling cycles of the frontend on each branch. While this technique is well established for hiding the latency of larger predictors, it may not be possible to use it effectively beyond a certain size. Previous work has found that a larger, slower predictor can be outperformed by a smaller and faster variant, as the increased accuracy of the predictions was offset by the additional latency costs [8]. Because overriding still requires all predictors to be accessed every time a branch is encountered, such overly large predictors would therefore hit scaling limits even in an overriding configuration, while also drastically increasing energy usage.

A recent design, "The Last-Level Branch Predictor" [18], approaches the problem from a different angle, aiming to address the limitations of previous hierarchical approaches, like paging. It extends an unmodified state-of-the-art TAGE-SC-L [19] predictor with a large backing storage, and leverages the fingerprint of the previous control flow events (such as function calls, returns, and jumps) to accurately identify the relevant patterns for predicting upcoming conditional branches. By exploiting the context locality of conditional branches, instead of the problematic spatial locality, the design enables prefetching only the necessary history into a small and fast buffer. During a prediction, this buffer can then be accessed in a single cycle, improving the accuracy of an underlying TAGE-SC-L [19] predictor without increasing its latency.

In this work, we want to understand the impact of latency on branch predictors, answering the following research questions:

- How much impact does branch prediction latency have on overall system performance?
- Can overriding branch prediction, which is currently used in many commercial processors, narrow the gap by hiding the latency of larger predictors? What are the scaling limits of this approach?
- "The Last-Level Branch Predictor" [18] is a new microarchitectural approach to scale the branch predictor without increasing its latency. Can it close the gap and realize the full opportunity of high-capacity branch prediction?

To answer these questions, we use gem5 [12] [3], the most prominent openly available computer architecture simulator. It models the full CPU pipeline, including associated caches and memory, and is highly configurable, supporting various architectures and branch predictors, making it a good fit for our evaluation.

What is, however, currently missing in gem5 is the simulation of access latency when making a prediction. Instead, all predictions inside the frontend are made instantly, which precludes an effective analysis of the impact of predictor latency.

Therefore, in this work, we will first provide an implementation for modeling both the access latency of a single predictor, as further explained in section 3.1, and the behavior when using two predictors with overriding, detailed in section 3.2. Afterwards, we port the design of LLBP [18] to gem5, building on top of an existing prototype from prior work Whitaker [22] in section 3.3. Then, different predictor configurations are tested on a suite of modern server workloads, and the results are presented in chapter 4. Finally, we give an overview of related work in chapter 5 and conclude our findings in chapter 6.

2 Background

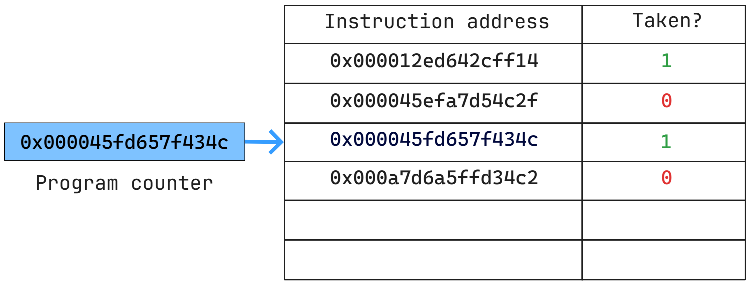
2.1 History-based branch prediction and TAGE

Conditional branch predictors have only one task: To determine whether a conditional branch instruction, e.g. `jne` or `jz`, will cause a jump in the control flow only based on its memory address. The exact instruction details are not known to the predictor, so it can only refer to its records from past occurrences of the same address. The only additional information forwarded to the predictor later on is the actual outcome of the branch, allowing it to correct its memory.

A simple example would be a predictor that has a table storing a mapping between memory addresses and a boolean value which is set to true (1) when the branch was taken last time, and false (0) otherwise. It then predicts the same result every time it encounters the same branch again with the entry in the table and updates its records once the actual outcome is known.

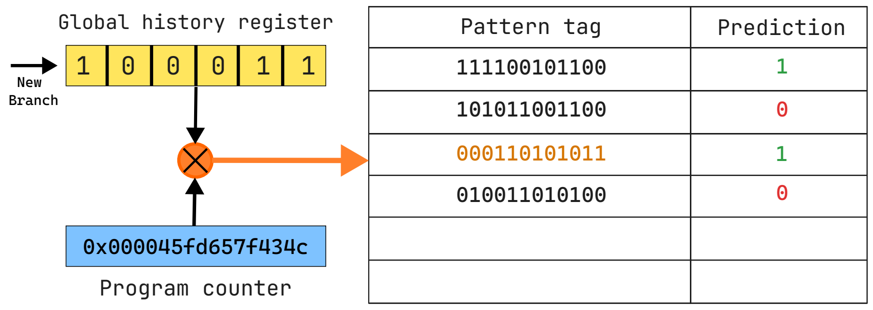
Such a predictor would already be capable of correctly predicting all directions of the branch at the end of a loop: `for(int i = 0; i < 100; ++i)`, except for the first one, where the branch is encountered for the first time, and the last one, where the behavior changes as the loop is complete. For more complicated branches, however, this simple predictor will perform poorly.

During the early development of branch predictors, it became apparent very fast that the most effective strategy to determine the next outcome of a branch is to learn its **correlation with the previous conditional branches**. In most programs, the direction



Instruction address	Taken?
0x000012ed642cff14	1
0x000045efa7d54c2f	0
0x000045fd657f434c	1
0x000a7d6a5ffd34c2	0

Figure 2.1: Simple predictor table



Pattern tag	Prediction
111100101100	1
101011001100	0
000110101011	1
010011010100	0

Figure 2.2: Global history predictor table

of a branch is heavily correlated with the previous conditional control flow and can be predicted by it. This strategy is known as **global history prediction** and is the base for most state-of-the-art predictors, like TAGE-SC-L [19].

In a practical sense, this approach could be implemented with a simple register of size n, which is shifted right every time a branch is encountered. If the branch is taken, a 1 is shifted in, 0 otherwise. This automatically leads to the oldest value on the very right to be shifted out and forgotten. The value of the register at any point in time now contains a fingerprint of the control flow of the previous n conditional branches; it is now a **global history register (GHR)**.

Returning to our simple example from earlier, we can now utilize this register to enhance the accuracy of our predictor significantly. On every received update about the outcome of a branch, we compute our table entry as usual, but instead of using the memory address of the branch directly, we calculate the **pattern tag** of the branch with an XOR operation between the memory address and the current value of our global history register (pattern tag $=$ addr $\oplus$ global history register) and store the result in our table together with the boolean from before. Now, the information about the current branch is interleaved with the previous control flow, and the same branch will be treated as a different entry if it is encountered after a different control flow.

Taking this simple idea further, the TAgged GEometric length predictor (TAGE) [20] uses not just one length n for its pattern tags, but instead geometrically increasing lengths $geo(n)$, with a separate storage table for each length.

When a prediction is made, TAGE searches each table for a match with a separate pattern tag, generated from the global history register together with the branch address. As every table contains patterns of different length, each tag will only consider a partial amount of the global history register. Once all tables are searched, the prediction will always be provided by the longest matching pattern, so the one which contains the most history. As a fallback, an untagged table without history information, similar to the simple table in Figure 2.1, is used when no match can be found with any of the history tables.

This has the advantage that branches that are easy to predict, and therefore do not require much history, can be stored in the shorter tables, while hard-to-predict branches that behave very differently depending on the previous control flow can be stored in the longer tables. To achieve this sorting in storage, TAGE will allocate the following longer pattern $geo(n+1)$ if a branch was mispredicted with a pattern of length $geo(n)$ to increase the amount of information about the previous control flow. Likewise, when the branch was correctly predicted with a pattern of length $geo(n)$, but would also be correctly predicted with a shorter table $geo(n-1)$, the longer entry will be marked for removal to save storage.

2.2 The Last-Level Branch Predictor

The Last-Level Branch Predictor [18] is a new hierarchical approach to branch predictor design that extends a baseline TAGE predictor [20] with a large backing storage that can store lots of additional history patterns. As this large storage has a long access latency, it cannot be directly used for predictions without encountering the same latency issues as a naive upscaling of TAGE. Instead, patterns are prefetched from the backing storage into a small core-side buffer. When a prediction is made, both the baseline predictor and the buffer are queried. If a match is found in the buffer that is longer or equal in size to the baseline prediction, it will override the baseline prediction to provide the final prediction instead of TAGE.

The key idea behind LLBP is to exploit the **control flow locality** that is present in branch prediction, similar to the nature of global history prediction itself. As it turns out, conditional branches not only exhibit a high correlation to previous conditional branches, which is used by TAGE in the form of global history, but they also correlate with the previous unconditional branches (jumps, function calls and returns) that form an execution **context** in which only specific patterns are required for prediction.

The overview of LLBP is shown in Figure 2.4. The authors extend a state-of-the-art 64KiB TAGE-SC-L [19] predictor with four components: A large backing storage (itself named "LLBP"), a small core-side pattern buffer, the rolling context register (RCR), and the context directory (CD).

The RCR is a small register that computes different hash values. When an unconditional branch is detected in the instruction stream, two values are updated in the RCR: The **prefetch context id (PCID)**, which contains the fingerprint of the program context by hashing the last W unconditional branches, and the **current context id (CCID)**, which contains the fingerprint of the historic execution context by hashing W previous

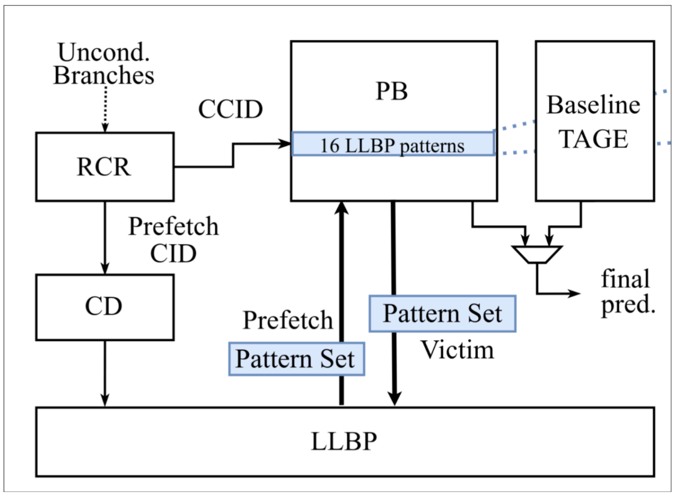


Figure 2.3: Overview of LLBP from the paper [18]

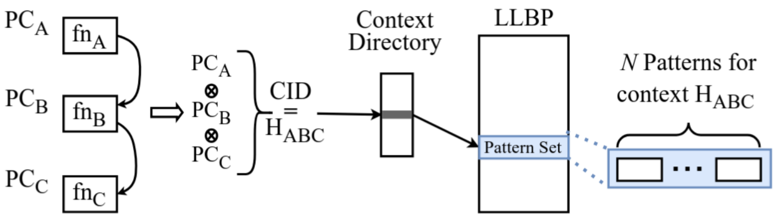


Figure 2.4: Functionality of the RCR [18]

unconditional branches *after skipping the D most recent unconditional branches*, so ignoring the recent history and looking at the past.

When a new pattern set is allocated in LLBP, it is associated with the CCID, so effectively with the fingerprint of a past execution context instead of the current context. Once this previous context is encountered again, the pattern buffer will already start fetching this pattern set from the backing storage, even though it is required only later, in the current context. By the time the pattern set is actually needed, prefetching is already completed, and the long access latency of the backing storage is completely hidden.

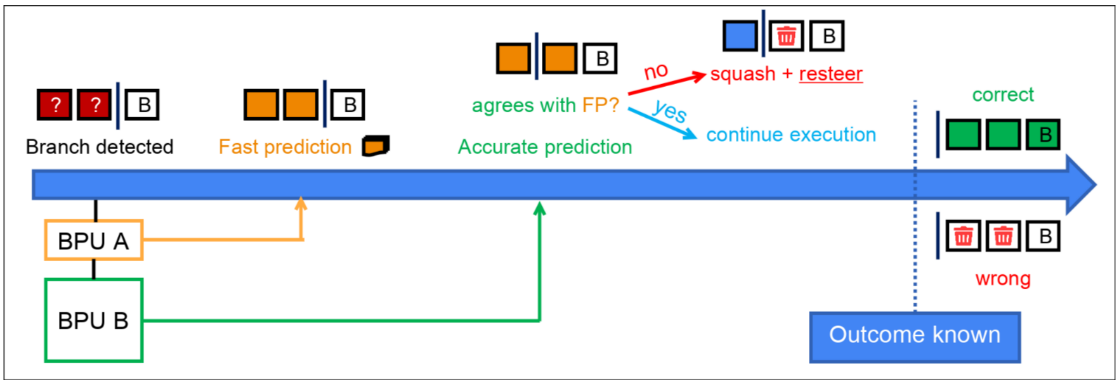


Figure 2.5: Overriding branch prediction

2.3 Overriding branch predictors

Overriding branch prediction is a common latency-hiding scheme used in today's processors. Instead of a single predictor, two or more predictors with increasing storage budgets and therefore increasing latency are included in the branch prediction unit.

As shown in Figure 2.5, when a branch is encountered, both the fast primary predictor and the slower secondary predictor begin computing their guesses simultaneously. While waiting for the secondary prediction, the pipeline will resume fetching instructions along the path obtained from the primary predictor. Once the secondary predictor returns its result, both predictions are compared. When a mismatch occurs, the fetched instructions are rolled back, and the pipeline will resteer to the new predicted path.

This has the advantage that the pipeline can start fetching instructions immediately after receiving the primary prediction, which is usually made by a very fast predictor with a single-cycle access latency. Therefore, the longer latency of the secondary predictor is effectively hidden and will not stall the pipeline, allowing for a higher latency tolerance compared to only using a single predictor.

2.4 The gem5 hardware simulator

The gem5 hardware simulator [12] [3] is widely regarded as the most comprehensive open-source simulation tool for computer systems and architecture research focused on the CPU pipeline and memory models.

We choose gem5 for this work over other trace-based simulators like Scarab [7] or ChampSim [6], due to its full-system simulation capabilities, making it possible to boot an unmodified Linux kernel to run workloads on a full operating system. System calls,

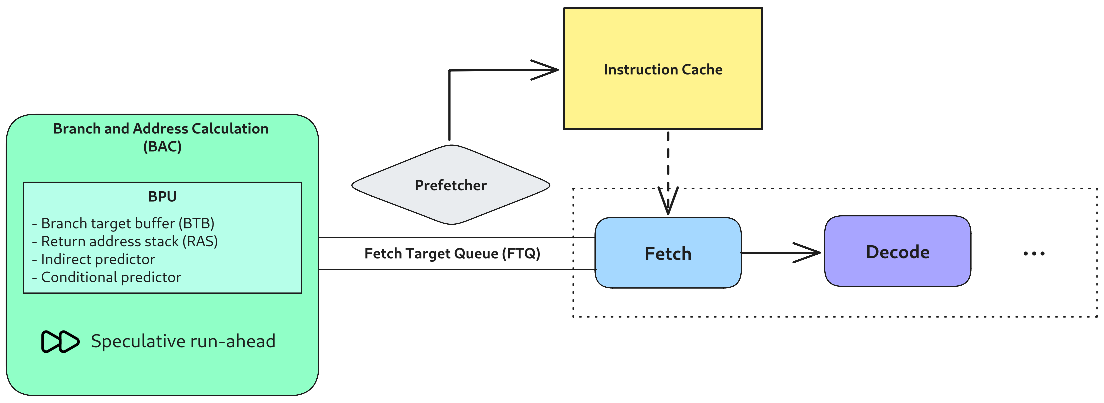


Figure 2.6: Architecture of the decoupled frontend

which are issued by the workload to perform I/O or other privileged operations, can therefore be handled directly inside the simulation and are accurately represented in the measurement results. In the context of modern server workloads, this feature is especially important, as they often contain a significant number of system calls due to their I/O-heavy execution profile.

To better represent the architecture of modern processors, we also make use of a gem5 fork that adds a decoupled frontend to the simulator [1]. Similar to many commercial CPUs today [13] [1], this design splits off the branch predictor from the fetch stage of the frontend. Instead of calling the branch predictor inside of fetch once a branch is encountered, and then stalling until the result is returned, the predictor runs ahead of the instruction stream, building a speculative path forward and inserting the following fetch targets into a queue, effectively buffering the upcoming instruction stream.

The architecture of the gem5 pipeline with a decoupled frontend is shown in Figure 2.6. The authors of gem5-fdp introduced a new stage to represent the independent part of the frontend, known as the branch and address calculation (BAC) stage. It is responsible for running ahead of the instruction stream with the branch prediction unit (BPU), which is a combination of the BTB, return address stack (RAS), indirect predictor for unconditional branches, and the conditional predictor. When a prediction is made, all the components get called in an order similar to real hardware, and a final predicted direction and target are returned. In its current implementation, the BPU and its components do not apply any latency for their predictions, but instead return the result immediately to the BAC stage. In section 3.1, this behavior is changed to enhance the realism of the model.

[1]https://github.com/dhschall/gem5-fdp/tree/fdp-develop

For each consecutive block of instructions that is contained between two branch instructions, only the first instruction address is inserted into the fetch target queue (FTQ) by the BAC stage, communicating the outline of the predicted path to the fetch stage, which can then prefetch the rest of the instructions in the block with a stride prefetcher.

To conduct our evaluation, the main component of interest is the conditional branch predictor, which can be configured by the user to use multiple existing designs, ranging from a simple 2 bit predictor to a full TAGE-SC-L implementation, or extended with new designs by inheriting from the base class 'ConditionalPredictor' and overriding the functions responsible for prediction, updating and speculative history update, which are called in the different phases of the BAC stage. This interface will later be used to evaluate The Last-Level Branch Predictor [18] in gem5.

3 Implementation

As described earlier, we utilize an experimental version of gem5 that features a decoupled frontend, similar to modern out-of-order processors. We extend this fork of gem5 with three contributions: Adding prediction latency to the conditional branch predictor, supporting two predictors that override each other, and providing The Last-Level Branch Predictor [18] as an additional conditional predictor component, extending previous work on an initial prototype [22] to reach a complete implementation.

3.1 Modeling predictor latency in gem5

In the gem5 code architecture, there are two types of components: Active and passive. Active components inherit from the 'SimObject' base class and expose a 'tick()' function to the core simulator, which is called once in every timestep of the simulation. These are often top-level objects, such as pipeline stages or processor cores, which communicate with each other over event queues in an asynchronous manner. In contrast, passive components such as the branch predictor or BTB are usually contained in an active component, which synchronously calls their available methods.

This has the consequence that it is not possible to delay the execution of the simulation, or introduce idle cycles for delays, inside of passive components. Instead, the information about the desired latency must be communicated to the enclosing active component, which can change its timing inside the 'tick()' function.

Currently, all components inside the branch predictor only return the direction and target of their prediction. To allow the conditional predictor to communicate the incurred latency to its next active parent, the branch and address calculation (BAC) stage, we modify the return type of the prediction function by adding a field for the prediction latency, measured in cycles.

The amount of prediction latency is configured statically in the gem5 configuration of each conditional predictor, but can then also be dynamically overridden by the predictor implementation for any predicted branch. This approach allows for a large amount of flexibility, as the predictor can model any possible latency behavior, for example, applying latency only on every second prediction.

To actually realize the latency in the decoupled frontend, we introduce a new type of stall into the BAC stage, which is automatically activated after a conditional branch prediction has happened in the same cycle, and will clear as soon as the specified latency has passed.

As the BAC stage is the decoupled part of the frontend, which is running ahead and adds its computed fetch targets to the FTQ, stalling it will only reduce the distance to the fetch stage, which runs unobstructed in the meantime and consumes elements from the queue until it is empty. This behavior closely matches the behavior of real hardware and should be able to capture the effects caused by prediction latency in modern frontends.

3.2 Overriding branch prediction in gem5

Building on the latency implementation of the previous section, we further extend the simulator with a model for overriding branch prediction. In addition to the primary conditional predictor present in the BPU, a secondary predictor can now be specified in the gem5 configuration, with its own prediction algorithm and latency characteristics.

To better understand the impact of overriding prediction with two predictors, we begin by examining the theoretical behavior within out-of-order processors.

Usually, the pipeline behavior for overriding prediction with two predictors P_f (fast, inaccurate) and P_s (slow, accurate) and their associated latencies L_f and L_s can be categorized into four cases:

1. Both predictors P_f and P_s have guessed correctly. This is the best case, as now the only direct latency cost is L_f. The arrival of the secondary prediction does not have any effect, and instructions can be directly fetched after L_f. We ignore for now the indirect cost that could occur when another branch has to be predicted directly after, and P_s is not ready yet.

2. Both predictors P_f and P_s have guessed wrong. This incurs the latency L_f when the frontend stalls to wait for the first prediction, but then also the penalty of a full pipeline flush when the misprediction is detected by the end of the pipeline. In this scenario, the performance is identical to that of a single-predictor configuration, as all fetched instructions since the branch will need to be squashed for both variants.

3. The predictors disagree, and the secondary predictor P_s is correct. In this case, the main advantage of an overriding scheme is obtained. As the delay L_s is usually still smaller than the number of cycles between the beginning of the pipeline and the point where a misprediction is detected, the pipeline is flushed earlier, and correct instructions can be fetched faster, which will outperform a single predictor configuration with only P_s.

4. The predictors disagree, and the secondary predictor P_s is wrong, meaning that the initial guess of P_f was correct. This case should be relatively rare, as the more accurate predictor P_s will almost never predict worse than the fast P_f. However, when it does happen, this scenario carries a very high penalty for the pipeline. After incurring the initial delay L_f, the pipeline fetches instructions on the correct path until P_s returns a different result. Then, the entire pipeline is flushed and fetching is restarted on the wrong path, finally obtaining the information that the first guess was correct all along, and flushing the pipeline again. As there are two flushes involved, the cost greatly exceeds a single predictor configuration with P_s.

Typically, a disagreement between the predictors would immediately cause a pipeline flush and a resteer of the frontend. However, due to the high complexity of the O3 CPU model of gem5, especially with the decoupled frontend, the required changes to implement an intermediate resteer of the pipeline during branch prediction would be highly invasive, create a wide range of edge cases, and take a much longer time frame to implement than what is available for this work.

Fortunately, it is possible to obtain very similar pipeline behavior to the cases described above with a much simpler approach. Instead of simulating the separate predictions after each other in different cycles, we can obtain both the fast and slow predictions **immediately** when encountering a branch. Now, there are only two cases:

- When the predictors agree with each other, we apply L_f cycles of latency to the decoupled frontend and return the agreed-upon prediction $P_s = P_f$, which correctly represents cases 1 and 2 from above.

- When the predictors disagree with each other, we instead apply a latency of L_s cycles to the frontend, but return the overriding prediction P_s instead of the fast prediction P_f. This model works because the stalling of the frontend due to latency produces a bubble of length L_s in the pipeline, which is very similar to the bubble that would be caused by flushing the pipeline up to the L_sth instruction. After the flush, the frontend would begin fetching along the path of the overriding prediction P_s, which is precisely what happens directly after the latency has passed in our model, covering the remaining cases 3 and 4 from above.

Looking at the practical implementation, we start by adding the overriding branch predictor as another passive component of type 'ConditionalPredictor' to the BPU, with its own latency settings and branch history object. Similar to the way latency is passed for the single predictor, the overriding predictor communicates its latency via the return type of its prediction method. However, before the latency data is passed to the BAC stage, the branch prediction unit compares the results of both the primary and secondary predictors to compute the final direction and applied latency, as described above.

3.3 The Last-Level Branch Predictor

To evaluate the performance of LLBP in gem5, it needs to be integrated into the simulation architecture for branch prediction. In prior work, a prototype was created for gem5 by Whitaker [22], which is used as the base structure for our work.

Our new LLBP conditional predictor component will replace the baseline predictor in the gem5 configuration. It contains a component of type TAGE-SC-L, which we will use internally to simulate a normal TAGE predictor by calling its methods for prediction and history updates ourselves, and computing our own prediction response and update operations. This way, the LLBP predictor maintains complete control over the order of operations and can insert arbitrary code before and after TAGE-SC-L is run.

In our implementation, all components, except for the context directory, are modeled similarly to the original paper, with some adjustments made to simplify the design while maintaining similar behavior. The context directory is omitted, and the metadata associated with each context is stored directly in the backing storage instead, as it does not affect the latency behavior of LLBP.

3.3.1 Backing storage

The high-capacity backing storage is specified as directly mapped in the original paper, which is represented with a hash map that maps the context ID to a pattern set and the 2-bit replacement counter normally found in the context directory. Similar to the original design, when a new context has to be allocated and there is no free storage available, the context with the lowest number of high-confidence patterns is replaced.

We note that, due to the removal of the 7-way associative context directory, the effective associativity of our implementation regarding the backing storage is unbounded, rather than 7-way associative, which can lead to a slight overestimation of the eviction algorithm's performance under high storage pressure.

However, since the replacement counter only carries 2 bits, the impact of this change will be relatively low. In most circumstances, the algorithm will have many equally weighted options for eviction, choosing a random context to evict from those, which will not significantly increase performance.

3.3.2 Rolling context register

Taking inspiration from the original code, the RCR is modeled as a list of previous branch addresses, which recomputes the current context ID and the prefetch context ID every time a matching branch is encountered. It can be configured with four parameters: Branch type T specifies the kind of branches that are included in the ID computation, where one can choose between conditional branches, unconditional branches, or both. The window size W dictates how many of the previous branches will be hashed, and in the case of the CCID, the skip distance D then specifies how many recent branches are skipped. Before hashing any address, it will be shifted right by C bits, which prevents overlapping between instructions in the lower bits, where Arm instructions frequently contain only zeroes.

Consistent with the original paper [18], the default values for the parameters are set to:

$$T = \text{unconditional}; W = 8; D = 4; C = 2$$

Internally, the RCR implementation will maintain the full memory address of at least the last $W + D$ branches (only those of type T) in the list and cache the computed IDs based on that list until it is updated.

Because we are targeting an out-of-order pipeline, additional considerations are needed when a misspeculation occurs. As the previously fetched instructions inside the pipeline are rolled back, this also has the consequence that all branches encountered on the incorrect path must be discarded, and the history of the past execution flow must be reset up to the mispredicted branch. To execute this reset, we store a snapshot of the current RCR list **inside the metadata of each branch**. When the predictor is informed that the branch was mispredicted and updates its history tables, we will also restore the RCR from the associated snapshot to correct its history.

3.3.3 Pattern buffer and prefetching

In the pattern buffer, up to 64 pattern sets can be stored, which are directly used for prediction. To facilitate faster querying, the buffer is modeled as 4-way associative in the original paper [18], which we will represent using a design similar to a hash map, where the context ID of each pattern set is mapped to a bucket with four elements. As the query time of the structure is not relevant for the simulated access, we model the buffer in this complex way only for purposes of the last-recently used (LRU) eviction algorithm. In an associative structure, any insertion of a pattern set can only evict pattern sets in the same bucket, and not the whole buffer, which influences the hit ratio of the buffer.

The pattern sets, which contain up to 16 patterns each, are also modeled with 4-way associativity, placing four consecutive history lengths in one bucket. Every pattern consists of the pattern tag, a signed 2-bit direction counter, and a valid flag. If available, invalid patterns (unused or evicted) are replaced first; otherwise, the least confident pattern in a set is replaced, indicated by the absolute prediction counter value.

With each update of the RCR, we schedule a prefetch from the backing storage into the pattern buffer by inserting the pattern set corresponding to the current PCID and storing the current clock cycle, `insertCycle`, in its metadata. In this way, we can later check how much time has passed since the prefetch was issued and filter out any pattern sets that should not yet be available for prediction. If no free slot is available in the buffer, the oldest pattern set is replaced.

If we detect that a pattern set is already contained in the pattern buffer, we do not issue a new prefetch for it.

3.3.4 Making a prediction

When the BPU requests a prediction from the LLBP implementation, we first obtain the result of the underlying predictor by calling the prediction method of TAGE-SC-L and storing the resulting direction and history length. Then, we check if the current context is known by looking up the current context ID (CCID) obtained from the RCR in the backing storage. If a match is found, we check whether the context was already prefetched by performing a lookup of the same CCID in the pattern buffer. When an entry exists, we then verify that the simulated prefetch latency has already passed, to ensure that the context was prefetched early enough into the pattern buffer to be eligible for the current prediction. To check for a match inside the contained pattern set, our implementation reuses the tag computation function from the baseline TAGE predictor, searching for the longest matching pattern inside the set. Finally, if a match is found, the history length is compared to the baseline prediction from above, and LLBP will provide the final prediction if its match is longer or equal to the TAGE prediction.

3.3.5 Prediction latency

To provide latency information to the frontend, our implementation of LLBP also utilizes the interface introduced in section 3.1. Since LLBP compares the prediction of the baseline predictor to the pattern buffer on every prediction, it incurs the full access latency of the baseline predictor.

The pattern buffer was estimated to be small enough for a single-cycle access speed by the authors of LLBP, which can be hidden behind the longer latency of TAGE, as both accesses can happen in parallel.

Therefore, we extract the access latency of the base predictor as returned from its prediction method, and return the LLBP prediction with the same latency.

4 Evaluation and discussion

4.1 Methodology

For the following experiments, we run a collection of server benchmarks from different sources, listed in Table 4.1, on the modified version of gem5. All workloads are warmed up for 100 000 000 instructions before measuring for 1 000 000 000 instructions with the built-in gem5 `stats.txt` output and the system configuration as seen in Table 4.2, unless stated otherwise in the following sections.

Benchmark	**Description**
Nodeapp	Node.js web server [17]
PHPWiki	PHP-FPM wiki application [17]
Cassandra, H2, H2O, Luindex, Lusearch, Spring, Tomcat	DaCapo Java benchmark suite [4] [17]

Table 4.1: Overview of the benchmarks

Architecture	gem5 arm ISA - full system mode
Core	3GHz, 6-way OoO, 512 ROB, 248/122 LQ/SQ
Memory	3GiB Dual Channel DDR4-2400MHz
Branch Predictor	TAGE-SC-L / LLBP
BTB	16K entry, 8-way associative
Cache	32KiB 8-way L1-I, 64KiB 16-way L1-D, 2MiB 16-way L2
Prefetcher	Instructions: FDIP + Tagged, Data: Stride

Table 4.2: System configuration of gem5

4.2 What is holding us back?

4.2.1 The storage opportunity

To understand the bottleneck that limited storage introduces into branch prediction, we begin by measuring the performance of TAGE-SC-L [19] with different storage budgets, without applying any latency.

For the state-of-the-art baseline, the existing TAGE-SC-L 64KiB predictor implementation from gem5 is used, with default settings. To simulate the upscaled versions of TAGE, we modify the parameters of the predictor (marked with *) to increase the number of entries each table can store, while leaving the auxiliary components, e.g. the loop predictor and statistical corrector, unchanged. Prior work [18] has shown that increasing the auxiliary components as well does not have a significant performance impact.

For the infinite-size version of TAGE-SC-L, we additionally increase the tag size to 20 bits for all tables, to reduce aliasing effects when a vast number of patterns is stored in each table.

Predictor	**Origin**	**Patterns per table**	**Pattern tag bits**
TSL-64KiB	gem5 built-in	2^{10}	8 (table 0-10), 12
TSL-128KiB	from 64KiB *	2^{11} *	8 (table 0-10), 12
TSL-256KiB	from 64KiB *	2^{12} *	8 (table 0-10), 12
TSL-512KiB	from 64KiB *	2^{13} *	8 (table 0-10), 12
TSL-Inf	from 64KiB *	2^{20} *	20 (all tables) *

Table 4.3: TAGE-SC-L Predictor Configurations

Looking at the accuracy results in Figure 4.1, we can observe, as expected, that higher amounts of storage directly result in a lower number of mispredictions across all benchmarks, corroborating previous work [10] [8].

The largest impact is achieved by doubling the storage capacity from 64KiB to 128KiB, reducing mispredictions by 0.3-47.7% (13.4% avg), with further increases from 128KiB to 256KiB and 256KiB to 512KiB providing an additional improvement of 8.6% and 6.9% on average.

In particular, the benchmarks `lusearch` and `nodeapp` appear to be responsible for this effect, as they experience a significant reduction in mispredictions of 47.7% and 22.4%, respectively, on the first size step from 64KiB to 128KiB. This suggests that the branch working set of these benchmarks is too large for a state-of-the-art 64KiB predictor, but can be almost entirely covered with a predictor of double the size. Therefore, increasing the capacity even further above 256KiB does not provide a similar improvement in predictor performance, as observed.

Providing an infinite amount of storage and larger 20-bit pattern tags to the predictor results in a combined 8.5% average reduction in mispredictions compared to the bounded 512KiB variant of TAGE-SC-L, indicating that most of the opportunity for additional storage can be found between 64KiB and 512KiB configurations.

Regarding the resulting system performance, we measure the improvement in instructions per cycle compared to the baseline TAGE-SC-L 64KiB predictor across all benchmarks for each predictor configuration, as shown in Figure 4.2.

The accuracy gains from increased storage generally lead to slightly better overall performance, with the 128KiB predictor outperforming the baseline by up to 1.89% (0.59% avg), increasing consistently with 256KiB of storage at 0.93% average and 512KiB at 1.12% average. As expected, the largest speedup is observed when using infinite storage, at 1.51% average.

Surprisingly, in the `h2o` benchmark, a predictor with 128KiB in size performs worse than the baseline, even though it experiences fewer mispredictions. We attribute this discrepancy to the size of the L1-I cache and its interaction with the decoupled frontend. When the branch predictor is highly accurate, the decoupled frontend can run significantly ahead of the fetch stage, inserting large amounts of prefetch targets into the FTQ. This can cause the eviction of cache lines from the L1-I cache that are still needed for upcoming instruction fetch operations, when the cache experiences high storage pressure, therefore decreasing overall performance. Similar behavior can be observed in the `cassandra` benchmark, where larger predictors generally result in reduced performance. When repeating the experiment for the specific benchmarks with a larger 64KiB L1-I cache, we did not observe this behavior

Take-away: Consistent with prior work on the topic [18] [10] [8], we observe a large reduction in mispredictions of up to 72.4% when providing the predictor with additional storage capacity, indicating that storage pressure is still a bottleneck for state-of-the-art predictors in modern server workloads. The effect of more accurate branch prediction on general system performance is also visible; however, it is lower than expected. A possible factor for this result will be discussed in the next section.

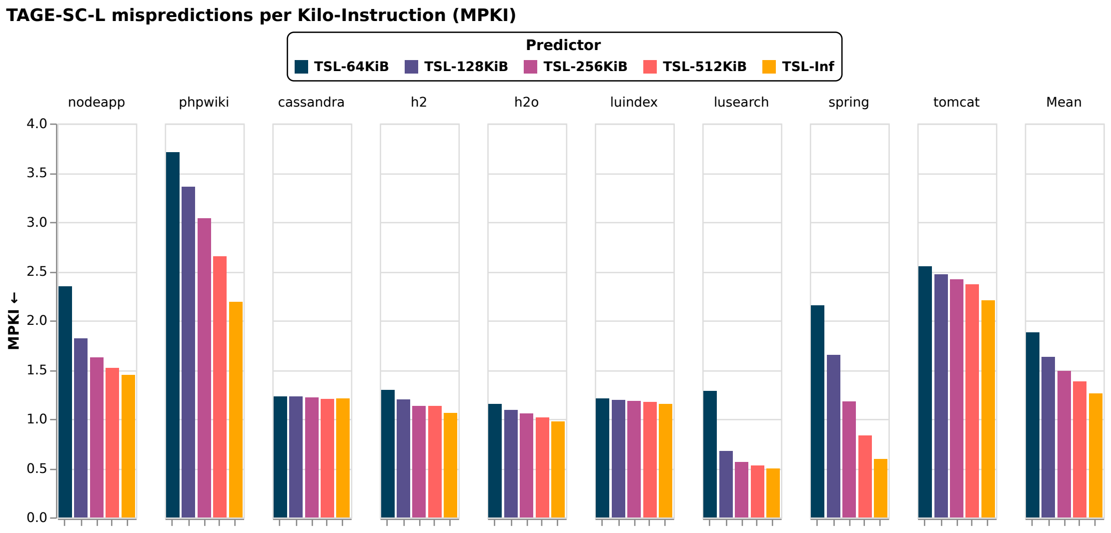


Figure 4.1: MPKI across server workloads with different storage budgets

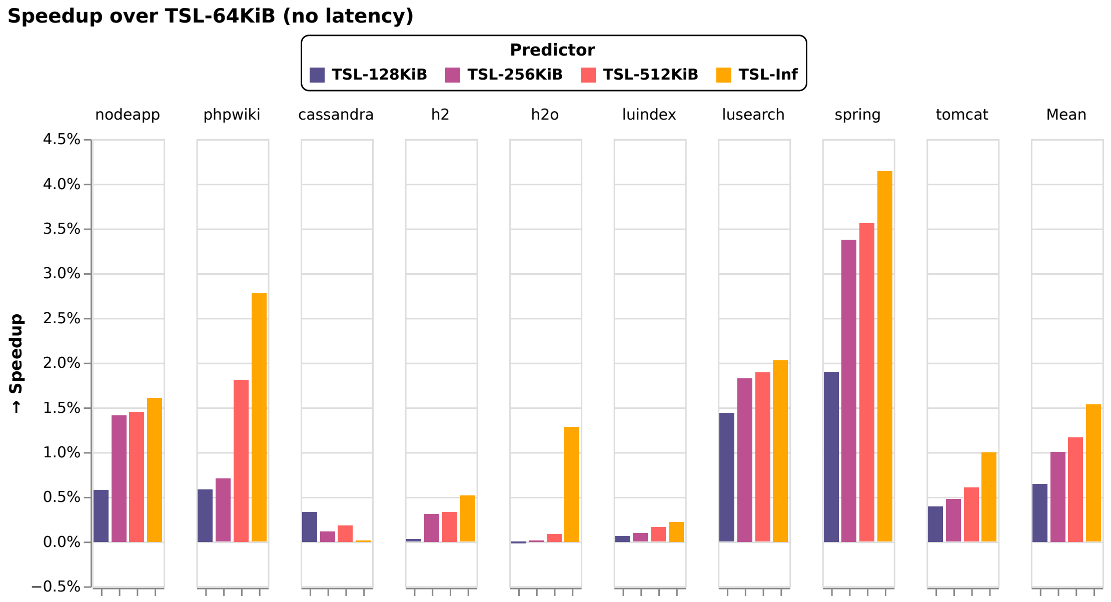


Figure 4.2: Speedup across server workloads with increasing storage budgets

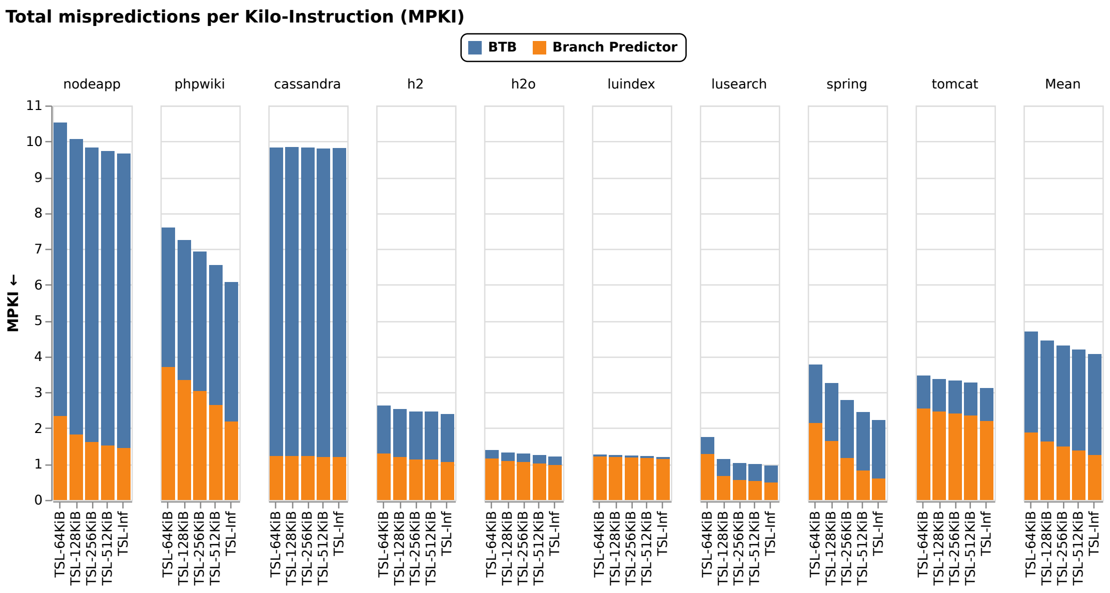


Figure 4.3: MPKI contributors across server workloads

4.2.2 The impact of large working sets

The branch prediction unit contains many different components, only one of which was scaled up in our experiments in the previous section. It is therefore possible that the overall processor performance is limited by other parts of the BPU in addition to the conditional predictor. When looking at the distribution of the total mispredictions between the branch predictor and BTB in Figure 4.3, this assumption is confirmed. With the baseline TAGE-SC-L 64KiB predictor, the branch predictor is responsible for only 40.1% of the mispredictions on average. In contrast, the other 59.9% are caused by incorrect or missing target addresses in the BTB, severely limiting the overall performance opportunity presented by improvements to the branch predictor. Naturally, this imbalance increases with the larger and more accurate predictors, reaching up to 69.1% on average for the infinite-size TSL variant.

It is generally known [2] [14] that a large instruction footprint, which is linked to a large branch working set, can overwhelm the storage capacity of the BTB, causing a high number of BTB misses due to evicted - but still useful - entries. To estimate the magnitude of this bottleneck, we measure the size of the branch working set in Figure 4.4 by counting the number of unique branch program counters that are sent to the BTB across the entire runtime of each benchmark, using the baseline TAGE-SC-L 64KiB predictor.

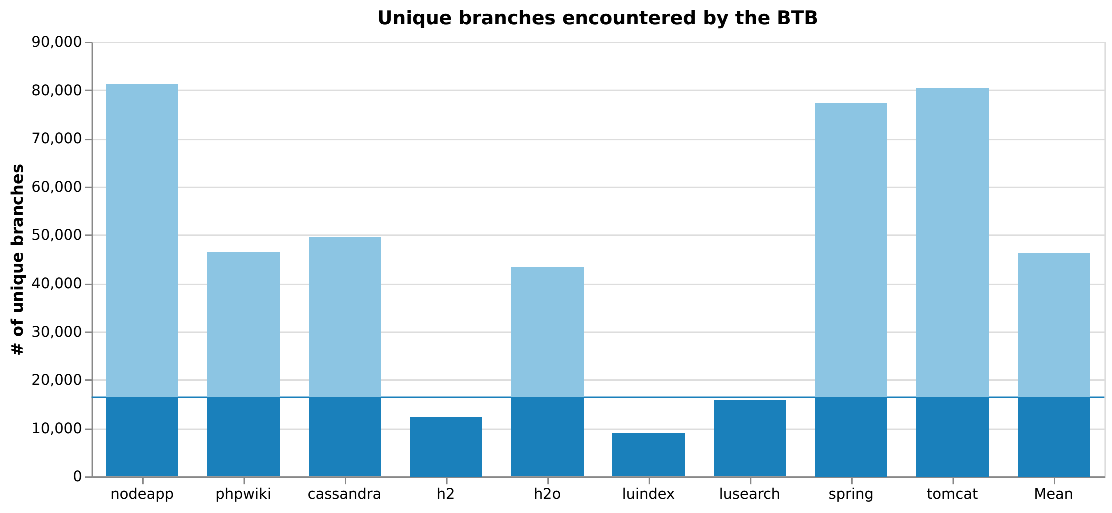


Figure 4.4: Number of unique branches encountered by the BTB

In the graph, the blue cutoff line represents the storage capacity of the BTB in our configuration, which is 16K-entries (16 384). As expected, many of the workloads overwhelm the BTB storage by a significant amount, requiring the tracking of 46 109 unique branches on average, which is 2.81x of the available capacity.

Although the results indicate that a storage bottleneck in the BTB storage is present, they still cannot fully explain the high miss rates observed previously. Especially when comparing the benchmarks `h2` and `h2o`, it becomes apparent that a large branch working set alone is not necessarily problematic for BTB performance in all circumstances. We assume this discorrelation to be caused by the access pattern of the working set across time, which heavily depends on the structure of the workload. Programs that have frequent and far-reaching jumps across the entire codebase will utilize very different parts of the branch working set within a short time, preventing the BTB from adapting to any bounded subset of branches and resulting in high miss rates. In contrast, if a workload first heavily uses branches from one code region and then jumps to another bounded code region, the BTB can store most of the necessary branches for the active region, even though its storage capacity would not be sufficient for storing the whole branch set.

4.2.3 The size-latency divide

Before evaluating the performance improvements of latency-hiding schemes, we must first obtain a baseline that represents the impact of branch predictor latency on system performance. To facilitate a comparison between different predictor sizes and latencies,

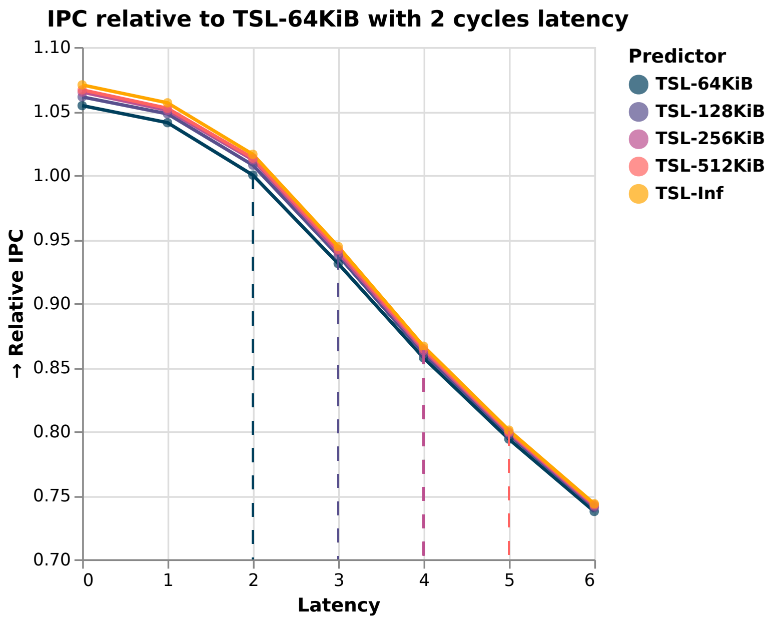


Figure 4.5: Relative IPC to TSL-64KiB with realistic latencies

we simulate each variant of TAGE-SC-L from before using the latency model introduced in section 3.1, configured to span 0-6 cycles of latency. All combinations of size and latency are then used to run the benchmark workloads from above.

The results are visible in Figure 4.5, where we measure the instructions per cycle for all predictor sizes relative to the TAGE-SC-L 64KiB variant, which serves as the state-of-the-art baseline. Relating to prior work [18], we assume a latency of 2 cycles to be realistic for this predictor, and extrapolate the access delay with a step increase of 1 cycle in latency for each increase by 2x in the capacity. These estimations are annotated with vertical dashed lines for each predictor. All results for a specific predictor that fall to the left of its respective dashed line are assumed to be impossible to achieve due to physical latency and energy constraints.

Three effects can be observed in the resulting data:

- Already with decreasing the latency of the baseline TAGE-SC-L 64KiB predictor by one cycle, a significant speedup of 3.6% is achieved, out of a total speedup of 5.9% for instant prediction. This indicates that reducing prediction latency for state-of-the-art predictors can already represent a significant performance opportunity.
- Every additional cycle of latency above two leads to massively degraded IPC

performance for all predictor sizes, with the largest mean loss of 7.64% being incurred when increasing the latency from 3 to 4 cycles. Even with the decoupled frontend, this indicates that realized latencies exceeding three cycles per prediction are not well tolerated by the pipeline.

- The performance difference between predictors of different sizes is small compared to the penalty applied by increased latency. Even with two cycles of latency, increasing the storage budget from 64KiB to infinite yields only a 1.63% speedup, while reducing latency by one cycle for the same 64KiB predictor improves performance by 4.09%.

Take-away: When directly exposing the decoupled front-end to longer prediction latencies, the overall system performance degrades very quickly, up to 7.64% with the increase of latency by a single cycle, nullifying any performance gains from increased prediction accuracy with larger storage budgets. Without latency hiding techniques, the pipeline's sensitivity to latency is generally much larger than the impact of higher prediction accuracy, making larger and slower predictors unattractive. Conversely, reducing the latency of the smaller state-of-the-art TAGE-SC-L 64KiB predictor to one cycle improved its performance by 4.09%, suggesting that latency is already a bottleneck with moderate storage capacity.

4.3 Overriding in the limit

As introduced before, one of the primary approaches for hiding the latency of larger and slower predictors is overriding. By providing a preliminary estimate with a small and fast predictor, the frontend can already begin fetching instructions, thereby reducing the effective latency.

We now compare the performance of an overriding configuration with the previous setup, where the BAC stage was directly exposed to increasing latencies of the TAGE-SC-L predictor.

As our fast primary predictor, we use the built-in gem5 `2bit_local` predictor, which models a simple table mapping the branch program counter to a signed direction counter with 2 bits, quite similar to the example predictor in Figure 2.1. We configure it with a capacity of 16K-entries, equating to approximately 4KiB of storage, and assume a single cycle of total prediction latency.

Using the behavior model introduced in section 3.2, we simulate the overriding of the 2-bit prediction with TAGE-SC-L in the size and latency configurations from the previous experiment: TAGE-SC-L from 64KiB up to 512KiB, with 0 to 6 cycles of latency.

For the experiments with zero cycles of TAGE-SC-L latency, we reuse the previous results, as overriding has no effect when the large predictor itself provides instant predictions.

4.3.1 Performance and latency

To compare the performance across all configurations with the baseline experiment, we plot again the relative IPC of each variant relative to the single TSL-64KiB predictor with two cycles of latency from section 4.2.3.

Across the board, introducing overriding into the architecture improved performance for all predictor sizes and latencies. Our baseline predictor, augmented with the single-cycle 2-bit primary predictor, now achieves 3.88% more IPC performance than the previous solo configuration at a 2-cycle latency. Also, the performance difference between the various predictor sizes is now much larger. It stays visible even with higher latencies applied, where doubling the capacity of the overriding TSL predictor to 128KiB at six cycles of latency results in a mean IPC increase of 0.61%, suggesting that the devastating effect of high latencies observed with the single-predictor configurations in section 4.2.3 is now significantly reduced.

As before, the dashed vertical lines represent our extrapolation of the lowest realistic latency for each predictor. We again consider all data points for a predictor located to the left of its respective vertical line to be unachievable. Additionally, the trend of IPC performance across each predictor's minimal realistic latency is represented by the dotted black line in the chart, marking the performance of TSL-64KiB at two cycles, TSL-128KiB at three cycles, etc. This metric shows the maximal performance we deem realistically obtainable across the different predictor sizes. When evaluating the performance of the overriding configuration, it is evident that doubling the storage capacity of the TSL-64KiB predictor, while also increasing its latency to three cycles, is now worthwhile, improving performance by 4.36% over the baseline, which is 0.54% more than the smaller 2-bit + TSL-64KiB configuration. However, when the storage capacity—and therefore latency—is increased even further, no additional benefit is obtained. Instead, using a TSL-256KiB with a 4-cycle latency in the overriding setup only achieves a 4.35% speedup compared to the baseline, degrading to 4.30% for the large TSL-512KiB configuration.

This performance ceiling shows the limitations of overriding branch prediction. Since the accuracy of the overriding predictions from TAGE-SC-L does not grow proportionally to its latency, the configuration still experiences diminishing returns, where a TSL-256KiB predictor with 4-cycle latency has a lower overriding performance than the faster but smaller TSL-128KiB predictor with 3-cycle latency.

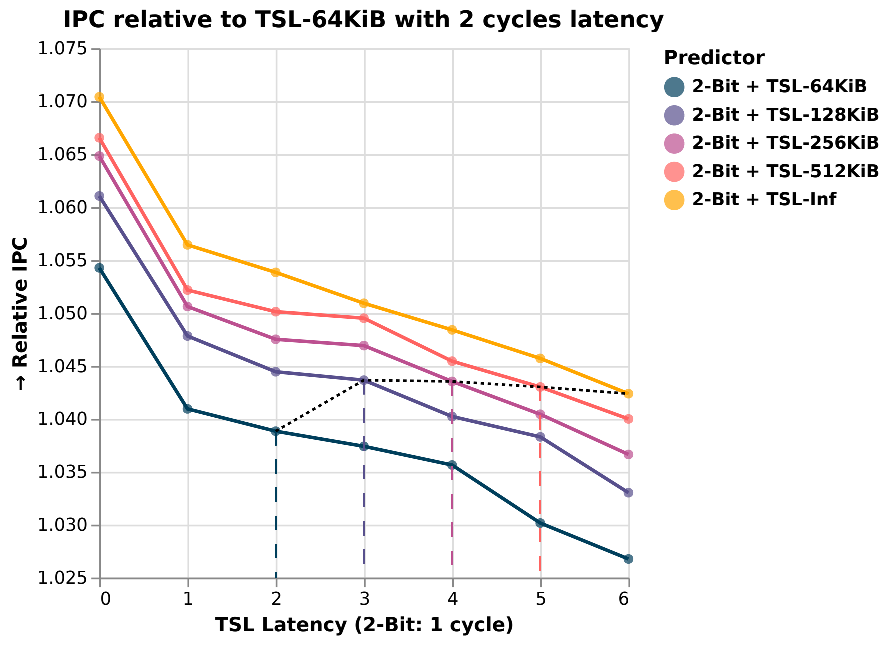


Figure 4.6: Relative IPC of overriding prediction with increasing TSL latency

When comparing the effective average latency for the previous TSL-64KiB solo predictor and the new overriding combination with the 2-bit predictor in Table 4.4, the reason for the stark improvement in latency tolerance becomes clear. When adding the fast 2-bit predictor to the setup, the effective average latency for each prediction reduces drastically, not exceeding two cycles per prediction, even when configuring a secondary TSL-64KiB predictor with a 6-cycle latency.

TSL-64KiB latency	Avg latency with 2-Bit	Improvement
1 cycle	1.000 cycles	0.0%
2 cycles	1.138 cycles	43.1%
3 cycles	1.271 cycles	57.6%
4 cycles	1.401 cycles	64.9%
5 cycles	1.527 cycles	69.4%
6 cycles	1.651 cycles	72.4%

Table 4.4: Measured average latency for baseline and overriding

4.3.2 Work distribution

When we again consider the properties of overriding prediction, as detailed before in section 3.2, the main driving factor behind effective latency reductions is the accuracy of the small primary predictor. Whenever the slower overriding predictor has to correct the first estimate of a branch, the latency cost increases drastically, as the pipeline must squash the previous instructions and resteer to a different path (represented by a higher prediction latency being applied in our model). Therefore, the primary predictor should have reasonably high accuracy on the performed workloads, to achieve a large latency benefit for overriding predictions. To understand the considerable improvement observed in the experiments, we track the number of committed conditional branches predicted correctly and incorrectly by each of the two predictors in Figure 4.7. In the plot, there are four possible outcomes for each conditional branch:

- **2-Bit correct** denotes the committed conditional branches which were correctly predicted by the small 2-Bit predictor without being overridden (i.e. the TSL predictor obtained an identical result). For each of these branches, the full TSL latency was hidden, therefore saving exactly L_s - L_f cycles per prediction.
- **Both wrong** applies when the small predictor mispredicted the branch, but the larger overriding predictor obtained the same incorrect result, therefore not overriding the initial prediction. Even though this outcome leads to an uncaught misprediction, the use of a small primary predictor still saves $L_s - L_f$ cycles of latency, as the slower predictor would have stalled the frontend longer, only to return the same incorrect result.
- **Bad override** is the case when the fast primary prediction would have been correct, but was overridden by the secondary predictor, forcing a misprediction of the branch. This case should be very rare, as it is expected that the larger predictor is much more accurate than the smaller primary predictor. Due to the override, we do not save any latency in this scenario compared to a solo TSL predictor.
- **Good override** of the larger secondary predictor means that the initial fast prediction was incorrect, and then corrected by the larger predictor, saving a misprediction compared to the less accurate primary predictor. As the pipeline would need to be flushed, this case also incurs the full latency L_s of the secondary predictor in our model.

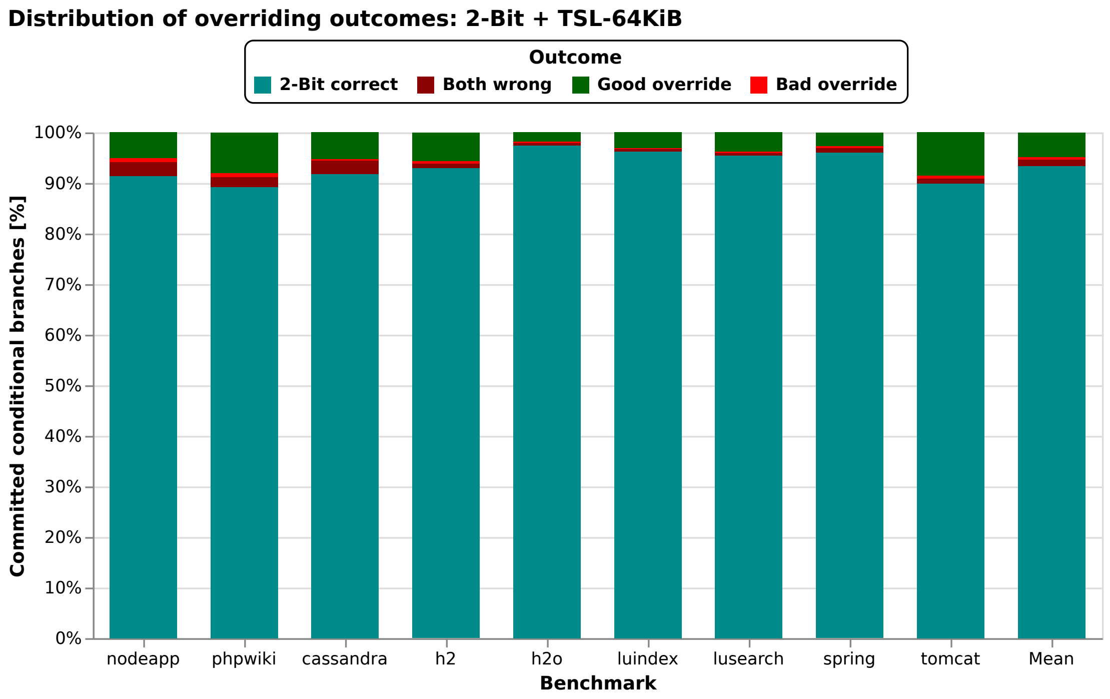


Figure 4.7: Distribution of overriding outcomes across benchmarks

It is apparent from the graph that the small 2-bit predictor has very high accuracy, making correct predictions for 89.18-97.45% (avg 93.36%) of committed conditional branches, without the intervention of the TAGE-SC-L 64KiB predictor. Additionally, in another 0.45-2.84% (avg 1.30%) of cases, both predictors were wrong about the direction of a branch, which is also still beneficial for the latency behavior overriding configuration. Therefore, 90.90-97.90% (avg 94.67%) of branches were predicted without an override, meaning they could be predicted identically to the current setup with only the small 2-bit predictor. For the remaining 2.10-9.10% (avg 5.33%) of committed conditional branches, the 64KiB TAGE-SC-L made a good override in 87.29-95.99% (avg 92.05%) of cases, preventing a misprediction from reaching the end of the pipeline.

This data explains the previously observed low effective latency and good latency tolerance. Since the 2-bit predictor manages to make the overwhelming majority of predictions alone, without being overridden by the larger and slower TSL-64KiB predictor, the full penalty of that predictor would only be incurred for 5.33% of branches on average, drastically reducing the average effective latency, as observed in Table 4.4.

The accuracy of the 2-bit predictor appears surprisingly high at first, compared to the much larger and more sophisticated TSL-64KiB state-of-the-art predictor. However, when calculating the results in reverse to obtain the corresponding MPKI, we get a value of 10.64, meeting our expectations. This shows the high sensitivity of MPKI as a metric, which is, however, well justified. Because the cost of every misprediction can be very high in large pipelines, wasting hundreds of cycles in some cases, even the misprediction of a small number of branches can incur a significant performance cost, which is reflected in the metric. One can conclude that in branch prediction, it is relatively straightforward to achieve an accuracy of around 90% using the simplest possible predictor, yet still leaving a significant performance gap, which becomes increasingly challenging to close when approaching 100% accuracy. Merely improving the accuracy from 93.36% with the base predictor to 98.28% with TSL-64KiB requires an increase of $\approx 16x$ in storage capacity and the introduction of very complex hashing and interleaving algorithms. It is therefore not surprising that further improvements from the current state-of-the-art predictors are increasingly difficult to obtain.

When comparing the different benchmarks, it also becomes clear that the performance of a simple 2-bit branch predictor is very workload-dependent. Some benchmarks, like `tomcat` and `phpwiki` show a large percentage of overrides in general, above 8%, and also a large amount of correct overrides, indicating that their working set contains many branches whose behavior is too difficult to predict for the simple 2-bit predictor, but which can be captured consistently with the global history structures of TAGE-SC-L. In other workloads, like `h2o`, `luindex`, and `spring`, the 2-bit predictor performs much better, resulting in fewer overrides being performed, reaching less than 2.1% in the case of `h2o`, which results in a higher efficiency for the latency behavior of the entire overriding configuration.

Take-away: Adding a 16K-entry 2-bit predictor with single-cycle latency to TAGE-SC-L in an overriding configuration drastically reduces the effective latency of the branch predictor, especially for larger and slower variants of TAGE-SC-L, by up to 72.4%. Consequently, this results in improved IPC performance, where even the moderately sized state-of-the-art TSL-64KiB with two cycles of latency outperforms the baseline by 3.88% on average. These stark improvements can be primarily attributed to the high accuracy of the fast 2-bit predictor, which reduces latency to a single cycle for 94.67% of committed conditional branches by predicting the same direction as TAGE-SC-L. However, when comparing the results across predictor sizes with increasing latency, it becomes clear that diminishing returns are still present. The overriding configuration reaches a performance ceiling of 4.36% IPC speedup with the TSL-128KiB predictor when assuming a minimal realistic latency of 3 cycles. Increasing the storage capacity—and therefore also latency—further does not yield additional improvements, indicating that simply upscaling the branch predictor does not improve performance beyond a certain point, which was also found by previous work [8].

4.4 The Last-Level Branch Predictor

As introduced earlier, The Last-Level Branch Predictor [18] is a new microarchitectural approach to hierarchical branch prediction, aiming to bridge the performance gap between accuracy and latency. After previously completing the gem5 implementation of LLBP in section 3.3, we will now evaluate the performance of the design on the various server workloads to compare it with the overriding configuration from the previous section. Because LLBP is designed to augment an existing TAGE-SC-L 64KiB predictor, providing additional storage and overriding its prediction when matching a pattern with an equal or longer history, it primarily influences the accuracy of the resulting predictions.

Therefore, we first evaluate its accuracy and the impact on overall system performance without any latency, using the same methodology as for the previous size comparison in section 4.2.1.

We configure LLBP with the parameters shown in Table 4.5, and try to match the settings of the original paper [18] as closely as possible.

Pattern buffer (PB)	64-entry, 4-way associative, LRU eviction
Pattern sets	16-entry, 4-way associative, 13-bit pattern tags
Rolling context register (RCR)	T = unconditional, W = 8, D = 4, C = 2, 14-bit tags
Backing storage (LLBP)	14K-entry, directly mapped [1] lowest-confidence-first eviction
Base predictor	TSL-64KiB

Table 4.5: Parameter settings of LLBP

4.4.1 Performance in modern workloads

To isolate the accuracy of the predictor from the rest of the results, we first measure the conditional mispredictions per Kilo-instruction, comparing the data from LLBP to the previous TAGE-SC-L configurations of different sizes in Figure 4.8. Because the total available capacity of the backing storage exceeds 500KiB, a significant reduction in mispredictions is expected.

Surprisingly, on the chosen workloads, LLBP achieves only a minimal average MPKI reduction of 0.07%, which is less than 5.3% of the improvement obtained by increasing the storage of TSL from 64KiB to 128KiB. While the design was capable of reducing mispredictions by up to 5.6% for some benchmarks, it did not manage to do so

[1]We do not model the 7-way effective associativity of the backing storage, see section 3.3.1 for details

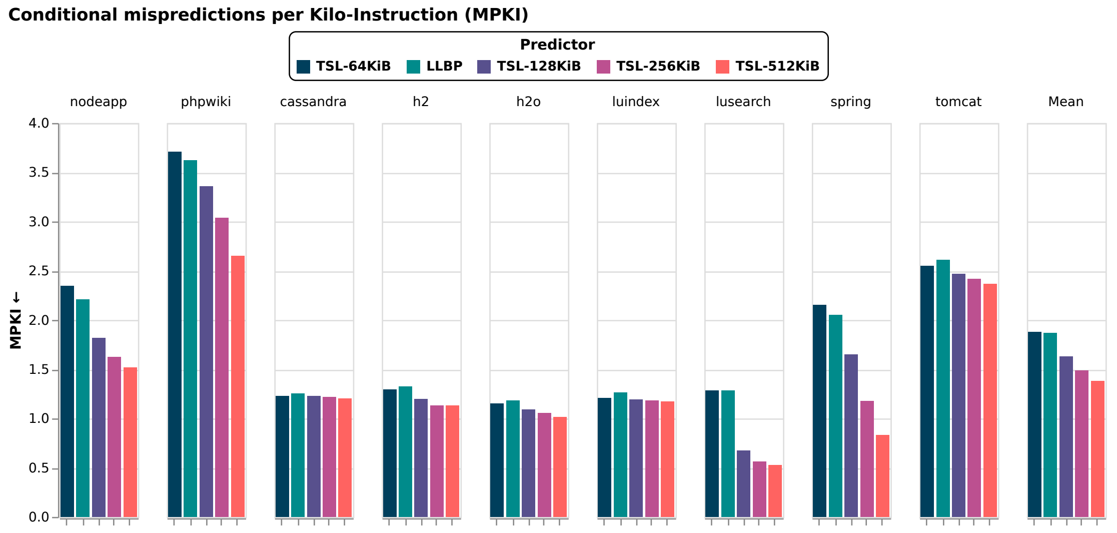


Figure 4.8: MPKI across server workloads with different predictors

consistently. In the workloads `tomcat`, `h2`, and `h2o`, LLBP even increased the number of mispredictions by up to 4.6%, indicating that the introduced components perform worse than the baseline predictor in certain scenarios.

To better understand the cause behind this unexpected result, we analyze the distribution of prediction accuracy across the components of the new setup. Unlike before, the analysis results do not permit any conclusions about the behavior of prediction latency. As detailed in section 3.3.5, LLBP has no impact on the prediction latency in its standard configuration, as it cannot decide on an override without waiting for the result of the primary predictor. Furthermore, the semantics of the different possible outcomes have changed, as the roles of the predictors are now reversed. LLBP will override TAGE-SC-L whenever it matches on a pattern in the pattern buffer with equal or longer history length, instead of TAGE-SC-L overriding a small predictor. Therefore, the following cases should be considered:

- **Both correct** and **Both wrong** both leave the accuracy and latency characteristics unchanged. Even though LLBP issued an override in these cases, the predicted direction was identical to the main TAGE-SC-L predictor, having no effect.
- **Good override** applies when LLBP provided the final prediction instead of TAGE-SC-L, which changed the predicted direction and was ultimately correct, meaning that TAGE-SC-L alone would have mispredicted the branch. This improves accuracy compared to a single-predictor configuration, while maintaining the same latency.

- **Bad overrides** degrade the overall accuracy, as TAGE-SC-L would have correctly predicted the branch, but was overridden by LLBP, which then made a misprediction. As in all other cases, the prediction latency remains unchanged.

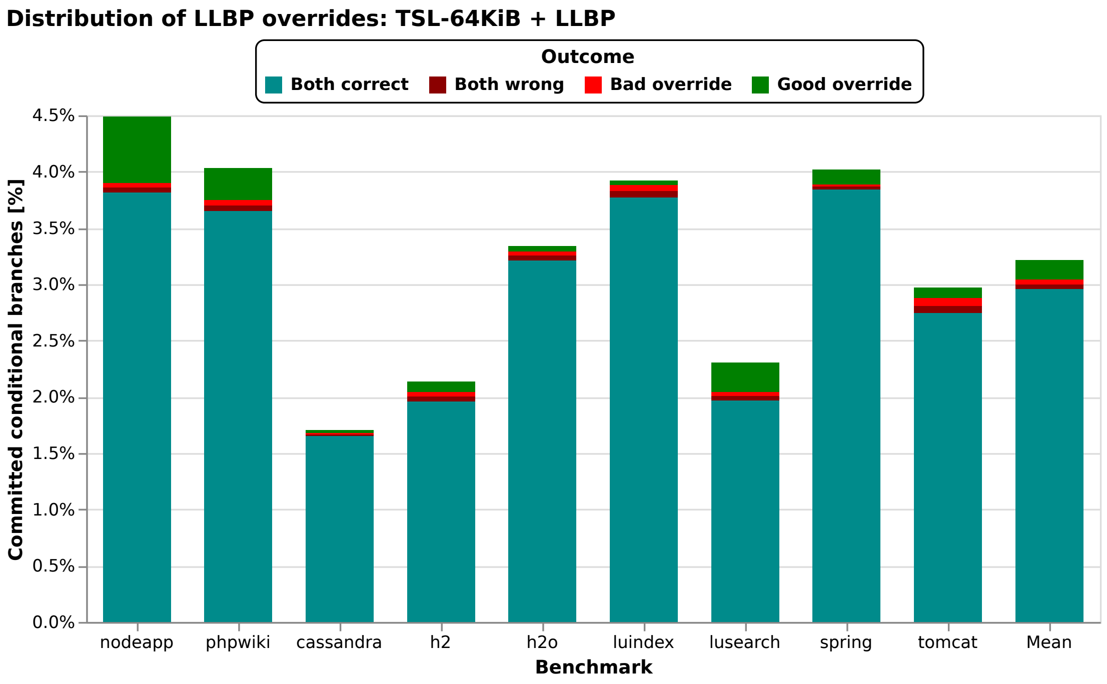


Figure 4.9: Distribution of LLBP overrides across benchmarks

In the resulting graph in Figure 4.9, we plot only the cases where LLBP has overridden the main predictor, which represent 1.7-4.5% (avg 3.2%) of all committed conditional branches. In the other 95.5-98.3% (avg 96.8%) of cases, the main predictor was the final provider of the prediction, without the influence of LLBP. Overall, it is evident that most of the overrides also did not change the outcome of the final prediction, as TSL-64KiB would have provided an identical guess to the LLBP override in 86.0-97.8% (avg 93.6%) of cases.

When focusing on the accuracy impact of the remaining 0.04-0.63% (avg 0.21%) of committed conditional branches, where the override of LLBP was different from the base prediction (denoted "unique overrides" from now on), the accuracy of the predictions ranged from 39.5% to 93.8%, with an average accuracy of 71.8%. This indicates that LLBP generally had a slightly positive impact on total prediction accuracy, resulting in a net reduction in misprediction of 0.13% on average.

This result closely matches the previously obtained number, which showed a 0.07%

average reduction in MPKI, where the remaining difference can likely be attributed to slight behavioral changes in the update methods of TSL-64KiB due to its integration into LLBP.

When comparing the results for different benchmarks, it becomes clear that the design does not perform well in many workloads, providing an increased number of bad overrides, most notably in `h2o`, and even reaching accuracy levels below 40% for unique overrides in the case of `luindex`, degrading performance.

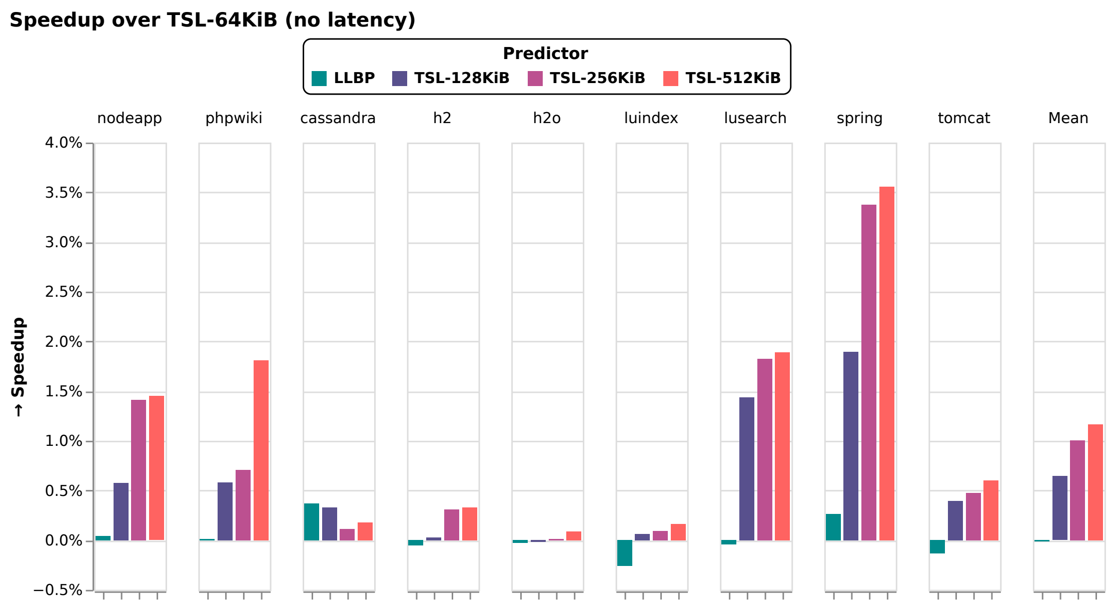


Figure 4.10: Speedup over TSL-64KiB across server workloads

Measuring the resulting IPC performance in Figure 4.10 as the percentage speedup over TSL-64KiB, we consequently observe very mixed results, where LLBP generally performs better than its baseline for the workloads in which the design also achieved improved MPKI performance, with a speedup of up to 0.26% in the case of `spring`. Similarly, in the benchmarks where LLBP suffered an increase in MPKI compared to TSL-64KiB, it also generally decreased the IPC performance relative to TSL-64KiB, resulting in a slowdown of up to 0.26%.

In summary, although LLBP generally reduced conditional mispredictions by 0.07% on average, it underperformed the baseline TSL-64KiB predictor by 0.01% on average. In addition to this surprising result, we also again observed discorrelated performance behavior in the `cassandra` workload, where an increase in MPKI by 1.72% resulted in a speedup of 0.37%, which is similar to the discrepancies that were present in the previous experiments with different sizes of TAGE-SC-L from section 4.2.1.

We note that the obtained accuracy and performance results are inconsistent with the previous work on LLBP [18], where the authors observed a much higher average accuracy for overrides, exceeding 90%, and also a larger number of overrides in general. The possible reasons for these differences and the limitations of our experimental setup are explained in section 4.4.3.

4.4.2 Latency performance

Due to the overall subpar performance results of LLBP, resulting in an underperformance of its baseline, even without any latency applied, it is not surprising that it does not exhibit improved behavior under the influence of latency when compared to the single-predictor configuration from section 4.2.3. This result can be seen in Figure 4.11, where LLBP experiences similar degradation due to latency as its baseline TSL-64KiB predictor, while achieving 0.07% lower IPC on average.

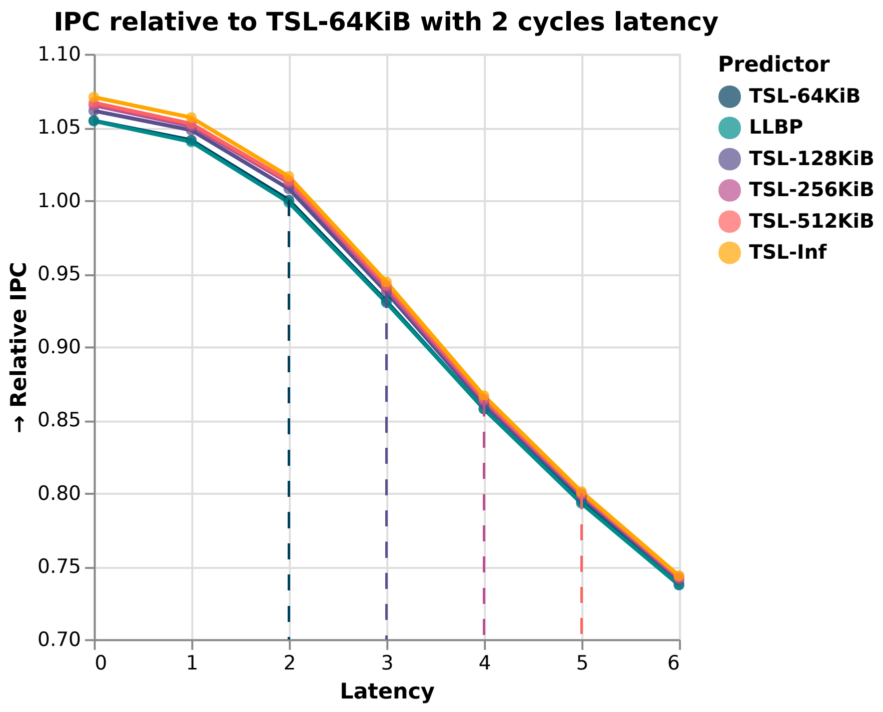


Figure 4.11: LLBP and TSL performance with increasing latency, relative IPC

4.4.3 Limitations of this work

In the experiments of the preceding sections, it became clear that the measured performance of LLBP is significantly lower than expected. Across the chosen benchmarks, the results vary considerably, with LLBP even underperforming its baseline predictor in some cases. Additionally, the obtained measurements differ substantially from the results of the evaluation that was performed in the original paper [18], where the authors found a 0.63% speedup across various server workloads.

The performance characteristics of LLBP are influenced by a diverse set of configurable parameters that determine the behavior of various components, including the rolling context register (RCR) and the pattern buffer.

Most notably, the RCR prefetch distance W, which determines the number of previous unconditional branches that are considered when computing the current context ID (CCID) and the prefetch context ID (PCID), impacts the number of different contexts that are allocated inside LLBP, which directly influences the storage pressure and aliasing effects in each context.

In the evaluation from the original paper, these parameters were adjusted for the specific benchmarks in use through empirical studies.

While several experiments were conducted with varying parameters of W during our implementation of LLBP in gem5, and a substantial performance difference was observed across all benchmarks, the results were not conclusive, as the differences were very inconsistent. While some of the benchmarks experienced improvements for certain values, others degraded, making it challenging to determine a single optimal value for conducting our experiments.

Therefore, due to the limited time frame available for this thesis, we did not perform empirical studies on the parameters of LLBP for the chosen set of benchmarks, and instead used similar settings as detailed in the original paper [18] wherever possible, which are listed in Table 4.5. However, in future work, it would be of interest to expand the analysis conducted in this work with an additional evaluation across many different configurations of LLBP, to fully explore the performance opportunities of the design.

5 Related work

Analysis of predictor latency: Jiménez, Keckler, and C. Lin [9] evaluated different predictors under the impact of latency, noting the tradeoff between accuracy and latency, and simulated various latency scenarios for the storage access time with smaller and larger feature sizes. Additionally, they found that overriding branch prediction was an effective tool in reducing predictor latencies. Their evaluation was conducted on the SPECInt 2000 benchmark suite [21], which considered feature sizes up to 35nm and is therefore no longer directly transferable to modern processors. However, the fundamental results remain valid and provide a general guideline for the performance of overriding prediction.

L. Cai, Deshmukh, and Patt [5] evaluated the performance of an overriding predictor consisting of a small 1K-entry 2-bit fast predictor and TAGE configured with different amounts of latency, and presented a novel technique for the energy-efficient implementation of ahead prediction, where predictions are made for future branches, hiding the long access latency of larger TAGE variants.

Latency-mitigation techniques: Jimenez [8] recognizes prediction latency as a key performance limiter and proposes a modified version of the gshare predictor, which pipelines the access to its history tables, reducing its latency to a single cycle. Due to its design, the approach is limited to predictors with a simple structure.

Schall, Sandberg, and Grot [18] introduced "The Last-Level Branch Predictor", a new microarchitectural design for an effective predictor storage hierarchy based on a context-sensitive prefetcher, which was re-implemented and evaluated in this thesis.

Other architectural approaches:

Pruett and Patt [15] focus on improvements of the branch predictor for hard-to-predict data-dependent branches, which are not reliably predictable for history-based predictors like TAGE, and propose hardware extensions of the pipeline to precompute a reduced version of the instructions that determine the outcome of those branches, which is then used to correct the prediction of the main conditional predictor.

Khan, Ugur, Nathella, et al. [10] present *Whisper*, a new holistic architecture for reducing branch mispredictions in datacenter workloads. They utilize offline profiling to collect traces on the execution paths that are frequently mispredicted, and then insert hints into the workload binary, which are consumed by additional hardware extensions in the branch predictor to replace the guess of a standard history-based predictor.

6 Conclusion

In modern, deeply pipelined processors, the performance of the branch predictor is crucial for effective instruction fetching; however, even state-of-the-art predictors such as TAGE-SC-L still struggle with the large instruction footprint of modern server workloads, primarily due to a lack of storage capacity. But when increasing the size of the predictor history tables, access latency also increases, which can offset the accuracy gains obtained from the increased storage capacity and create a performance ceiling that can only be surpassed with latency mitigation techniques.

In this work, we first analyzed the performance opportunity of increased storage for the TAGE-SC-L predictor, using the gem5 system simulator to run a collection of server benchmarks with various predictor configurations.

We found that quadrupling the storage capacity of TAGE-SC-L from 64KiB to 512KiB results in 28.9% fewer mispredictions on average across modern server workloads. When no latency is involved, this results in a 1.12% average IPC speedup, confirming the assumption that storage remains a bottleneck for modern branch predictors.

To quantify the impact of predictor latency on overall performance, we then extended gem5 with the capability to simulate prediction latency. We repeated our experiments, using TAGE-SC-L with different sizes and simulated increasing amounts of prediction latency to the pipeline. As expected, the IPC performance is affected significantly, where even adding a single cycle of latency to a baseline 2-cycle TAGE-SC-L 64KiB predictor resulted in a performance drop of over 6%. Additionally, we found that no amount of additional storage can offset an increase in latency, which is consistent with prior work [8].

Having observed the drastic performance impact of prediction latency, we continued our evaluation by comparing two approaches to latency reduction in the branch predictor: Overriding branch prediction, which is used in many modern processors, and "The Last-Level Branch Predictor" [18], a novel design for hierarchical branch prediction that promises to increase the accuracy of an underlying TAGE-SC-L 64KiB predictor without additional latency. To conduct the evaluation of these two designs, we first implemented corresponding simulation models in gem5 and then repeated the previous experiments, comparing their performance and latency tolerance with the previous single-predictor baseline.

The overriding configuration, in which we combine a single-cycle 2-bit bimodal predictor containing 16K-entries with different sizes of the TAGE-SC-L predictor, substantially increases performance in high-latency settings and reduces the average latency per prediction by up to 72.4%. This suggests that the realistic use of higher TAGE-SC-L storage budgets is now possible, as the increased accuracy can offset the slight overall increase in latency up to a predictor size of 128KiB. Therefore, a 3-cycle TAGE-SC-L 128KiB predictor now outperforms a smaller but faster 2-cycle TAGE-SC-L 64KiB by 0.48% on average, when both are placed behind the single-cycle 2-bit predictor in the overriding configuration. Still, beyond 128KiB of storage, no additional benefit is obtained, limiting the maximum IPC improvement to 4.36% over the baseline.

In contrast to the promising results of the overriding technique, The Last-Level Branch Predictor (LLBP) performed slightly worse in IPC than the baseline TSL-64KiB predictor, even at no latency. Despite minimally reducing the average mispredictions compared to the baseline, the results were generally mixed across the chosen workloads, with some benchmarks experiencing up to 5.6% fewer mispredictions under the influence of LLBP, with others even seeing an increase in MPKI of up to 4.6%. Due to the low performance of LLBP, it did not manage to tolerate latency better than the baseline TSL-64KiB predictor, and could not surpass the performance ceiling of the overriding configuration.

The general behavior of LLBP appears to be inconsistent with the results obtained by the evaluation of the original paper [18] and varies considerably across different workloads, which we attribute to the design's sensitivity to the chosen parameters. In the original paper, the parameters for the various predictor components were adjusted to the specific set of evaluated benchmarks through empirical studies. Due to the limited timeframe of this work, we did not repeat this process, reusing the parameters instead.

A valuable direction for future research would be to broaden the current analysis by systematically assessing a wide range of LLBP configurations to fully explore the approach's potential when adapted to specific workloads.

Abbreviations

BPU	branch prediction unit
FTQ	fetch target queue
LRU	last-recently used
GHR	global history register
BTB	branch target buffer
RAS	return address stack
TAGE	TAgged GEometric length predictor
RCR	rolling context register
BAC	branch and address calculation

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] T. Asheim, B. Grot, and R. Kumar. "BTB-X: A Storage-Effective BTB Organization." In: *IEEE Computer Architecture Letters* 20.2 (2021), pp. 134–137. DOI: `10.1109/LCA.2021.3109945`.

[3] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. "The gem5 simulator." In: *SIGARCH Comput. Archit. News* 39.2 (Aug. 2011), pp. 1–7. ISSN: 0163-5964. DOI: `10.1145/2024716.2024718`.

[4] S. M. Blackburn, Z. Cai, R. Chen, X. Yang, J. Zhang, and J. Zigman. "Rethinking Java Performance Analysis." In: *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2025, Rotterdam, Netherlands, 30 March 2025 - 3 April 2025*. ACM, 2025. DOI: `10.1145/3669940.3707217`.

[5] L. Cai, A. Deshmukh, and Y. Patt. "Enabling Ahead Prediction with Practical Energy Constraints." In: *Proceedings of the 52nd Annual International Symposium on Computer Architecture*. ISCA '25. New York, NY, USA: Association for Computing Machinery, 2025, pp. 559–571. ISBN: 9798400712616. DOI: `10.1145/3695053.3730998`.

[6] N. Gober, G. Chacon, L. Wang, P. V. Gratz, D. A. Jimenez, E. Teran, S. Pugsley, and J. Kim. *The Championship Simulator: Architectural Simulation for Education and Competition*. 2022. arXiv: `2210.14324 [cs.AR]`.

[7] HPS Research Group. *Scarab*. 2022. URL: `https://github.com/hpsresearchgroup/scarab` (visited on 06/30/2025).

[8] D. Jimenez. "Reconsidering complex branch predictors." In: *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.* 2003, pp. 43–52. DOI: `10.1109/HPCA.2003.1183523`.

[9] D. A. Jiménez, S. W. Keckler, and C. Lin. "The impact of delay on the design of branch predictors." In: *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*. MICRO 33. Monterey, California, USA: Association for Computing Machinery, 2000, pp. 67–76. ISBN: 1581131968. DOI: `10.1145/360128.360137`.

[10] T. A. Khan, M. Ugur, K. Nathella, D. Sunwoo, H. Litz, D. A. Jiménez, and B. Kasikci. "Whisper: Profile-Guided Branch Misprediction Elimination for Data Center Applications." In: *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2022, pp. 19–34. DOI: `10.1109/MICRO56248.2022.00017`.

[11] C.-K. Lin and S. J. Tarsa. "Branch Prediction Is Not A Solved Problem: Measurements, Opportunities, and Future Directions." In: *2019 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, Nov. 2019, pp. 228–238. DOI: `10.1109/iiswc47752.2019.9042108`.

[12] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillon, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, C. Escuin, M. Fariborz, A. Farmahini-Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, A. Gutierrez, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, M. Moreto, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, W. Wang, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and É. F. Zulian. *The gem5 Simulator: Version 20.0+*. 2020. arXiv: `2007.03152 [cs.AR]`.

[13] 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`.

[14] A. Perais and R. Sheikh. "Branch Target Buffer Organizations." In: *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '23. Toronto, ON, Canada: Association for Computing Machinery, 2023, pp. 240–253. ISBN: 9798400703294. DOI: `10.1145/3613424.3623774`.

[15] S. Pruett and Y. Patt. “Branch Runahead: An Alternative to Branch Prediction for Impossible to Predict Branches.” In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO ’21. Virtual Event, Greece: Association for Computing Machinery, 2021, pp. 804–815. ISBN: 9781450385572. DOI: `10.1145/3466752.3480053`.

[16] M. Sadooghi-Alvandi, K. Aasaraai, and A. Moshovos. “Toward virtualizing branch direction prediction.” In: *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2012, pp. 455–460. DOI: `10.1109/DATE.2012.6176514`.

[17] D. Schall. *gem5 server benchmark suite*. 2025. URL: `https://github.com/dhschall/gem5-svr-bench` (visited on 06/30/2025).

[18] 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`.

[19] A. Seznec. “TAGE-SC-L Branch Predictors.” In: *Proceedings of the 4th Championship Branch Prediction*. Minneapolis, United States, June 2014.

[20] A. Seznec and P. Michaud. “A case for (partially) tagged geometric history length branch prediction.” In: *The Journal of Instruction-Level Parallelism* 8 (Feb. 2006), p. 23.

[21] Standard Performance Evaluation Corporation. *Spec CPU2000 benchmarks*. 2006. URL: `https://www.spec.org/osg/cpu2000/` (visited on 06/30/2025).

[22] C. Whitaker. *LLBP prototype code for gem5*. 2024. URL: `https://github.com/CaedenWhitaker/gem5/commit/6d15529d6cf852bf80da42ee678ce100c6074424` (visited on 07/03/2025).