Unified Frontend Prefetching

Mária Ďuračková
University of Edinburgh
Edinburgh, UK
maria.durackova@ed.ac.uk

1 Background

1.1 BPU's Importance in Instruction Supply

Today's software stacks have reached massive instruction footprints. These footprints far exceed typical front-end capacity limits. The bloated footprints overwhelm the branch predictor, branch target buffer, and instruction cache creating severe front-end bottlenecks [2]. Branch direction and target mispredictions trigger costly pipeline flushes. Instruction cache misses stall the processor for tens of cycles while fetching from lower cache levels. The importance of a steady instruction supply grows even more significant given the recent trend of cores becoming wider and bulkier (larger ROB, wider pipeline, larger BTB, caches etc.). A reliable instruction stream is critical for harnessing the massive instruction-level parallelism of modern cores and will become even more vital as processor designs grow increasingly aggressive.

The branch prediction unit (BPU) is crucial in instruction supply. It speculatively predicts the branch instruction outcomes – the branch direction using the branch predictor (BrP) and the branch targets using the branch target buffer (BTB), return address stack and indirect predictor. BPU's role in instruction supply is two-fold. Its primary role is to predict the branch outcomes and eliminate unnecessary pipeline flushes. The secondary role is in the Fetch Directed Instruction Prefetching (FDIP) [18, 19], where the BPU is decoupled from the instruction fetch unit (IFU) and allowed to run ahead of the IFU to issue prefetch requests based on the BPU's predictions. FDIP is an attractive instruction prefetching solution, mainly due to its near-zero capacity requirements and relatively high accuracy. The accuracy, however, is highly dependent on the BPU's accuracy, as a single branch misprediction invalidates all consecutive prefetch targets and potentially causes cache pollution by prefetching the wrong targets.

1.1.1 Branch Target Buffer. The Branch Target Buffer (BTB) is a component of the BPU, responsible for caching past predictions of target addresses to make predictions. BTB holds a **single entry** per branch. Any BTB miss causes the BPU to generate the wrong target address for all taken branches and makes FDIP unable to prefetch the correct instrution path. While numerous solutions have been proposed to enhance BTB accuracy - from hierarchical designs to prefetching mechanisms like [4, 13], combined BTB/I-Cache prefetchers such as [12, 14, 15], and many other efficiency enhancement techniques such as [1, 16, 23] - the effort to improve the BrP presents a distinct set of challenges.

1.1.2 Branch Prediction Challenges. The state-of-the-art branch predictor 64KB TAGE-SC-L[22] predicts a given branch's direction by partial pattern matching [17]. It identifies correlations by comparing the current global history of branch outcomes leading to a branch with the previously observed patterns of branch outcomes. A history of previous branch outcomes associated with a given branch instruction is referred to as a pattern. There can be **multiple patterns** associated with one branch.

While effective, TAGE faces significant challenges in achieving further improvements. Simply scaling up its structures proves counterproductive, as any accuracy gains are offset by increased latency costs. Another option is a hierarchical design of TAGE, which presents two major obstacles. First, TAGE's extensive use of hashing destroys the natural locality patterns of individual branches, making it difficult to efficiently partition predictor entries. Second, branch behaviour exhibits highly skewed pattern distributions across different PCs (some PCs have disproportionately many patterns associated with them), rendering straightforward per-PC paging approaches ineffective due to bandwidth constraints. Any viable hierarchical TAGE design must therefore address both the locality destruction from hashing and the challenge of handling asymmetric pattern distributions. [21]

Recent work introduced the Last Level Branch Predictor (LLBP) [21], a design tackling aforementioned challenges in making TAGE hierarchical. It consists of a large storage structure which organizes the TAGE's patterns into so-called contexts. A context is a series of recently committed unconditional branches (jumps, calls, returns), similar to a call graph. Each context is small enough (16 TAGE patterns) to be effectively prefetched into a smaller prediction structure ahead of time to make a timely and more accurate prediction.

1.2 How To Make the BPU Better?

Modern server workloads with massive instruction footprints overwhelm both the BrP and BTB, necessitating effective prefetching solutions for both structures. While separate prefetchers could address each component individually, this approach would incur unnecessary overhead in terms of storage, energy, and complexity. A unified prefetching mechanism that can simultaneously handle both BrP and BTB entries is therefore more attractive, particularly given the tight coupling between these components in the branch prediction unit.

The asymmetric nature of BrP and BTB entries makes it more logical to embed BTB prefetching into a BrP prefetcher rather than vice versa. While each branch has only one corresponding BTB entry, it can have multiple TAGE patterns associated with it, making pattern management more complex. The LLBP already solves the challenge of managing TAGE patterns through its context-based organization of TAGE patterns, making it a natural foundation for a unified prefetcher. By extending LLBP's contexts to include BTB entries, we can leverage its existing prefetch mechanism while maintaining its efficient handling of multiple patterns per branch.

YArch 2025, March 31, 2025, Rotterdam, Netherlands
2025. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Mária Ďuračková
University of Edinburgh
Edinburgh, UK
maria.durackova@ed.ac.uk

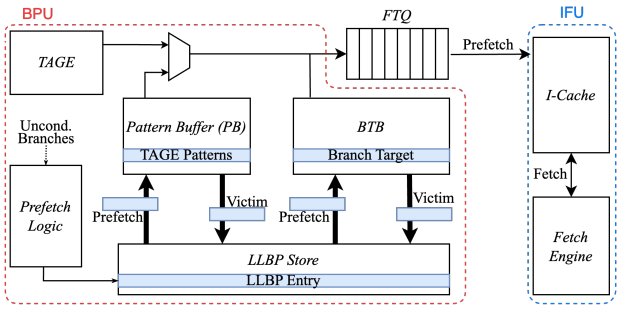

Figure 1: High-level Design of the Unified Frontend Prefetcher.

2 Unified Frontend Prefetching

We propose a unified frontend prefetching mechanism (UFP), illustrated in Figure 1, which simultaneously prefetches TAGE patterns, BTB entries, and instructions while sharing metadata storage. By extending LLBP to include BTB entries, UFP leverages LLBP's context-based design to enhance BPU accuracy through more precise branch prediction and target buffering. This improved BPU accuracy enables FDIP to generate more accurate speculative paths, leading to more effective instruction prefetching.

UFP extends the LLBP's design by embedding the BTB metadata to an LLBP entry. Therefore, one entry in the LLBP store will consist of TAGE patterns, and a BTB entry. The prefetch mechanism is triggered every time a new unconditional branch is committed. The unconditional branch's PC is inserted into the prefetching logic, which generates a hash to index the LLBP store and initiate the prefetch ([21] describes the prefetching logic).

UFP prefetches TAGE patterns into Patterns Buffer (PB), a structure which identifies the longest matching pattern of the current branch, and makes a prediction alongside TAGE. If both TAGE and PB make a prediction, the prediction with the longest history is chosen. While patterns are prefetched into PB, a BTB entry associated with the prefetch context is inserted into the BTB.

The rest of the mechanism is similar to any other FDIP design. The speculative address of the next basic block is generated by the BPU and pushed into the Fetch Target Queue (FTQ). FTQ operates as a FIFO where each entry represents a basic block. FTQ issues prefetches of cache lines associated with each FTQ entry into the L1-I cache.

The most scarce resource in the UFP architecture is bandwidth, therefore, we aim to minimise the size of an LLBP entry. We explore reducing the number of TAGE patterns in LLBP while preserving branch prediction accuracy. Additionally, given that a typical BTB entry is about 10 bytes long, we aim to lower the overhead by limiting each context to one BTB entry. However, our previous analysis revealed that the average number of BTB entries per context exceeds one. There are several options to solve the bandwidth problem.

Firstly, the BTB entry size can be reduced by storing only offsets for both the tag and target instead of full addresses, significantly lowering the storage overhead - an approach inspired by BTB-X [1]. Potentially allowing to store more than one BTB entry per context.

Secondly, the number of BTB entries per context can be reduced by optimizing how contexts are generated. We can control context specificity, by dynamically adjusting the number of unconditional branches used in context hash generation. Using more branch PCs in the hash creates more specific contexts, naturally distributing BTB entries across multiple contexts and reducing BTB entries per context.

Thirdly, BTB entries which are more likely to cause a long stall can be identified and prioritised when allocating the BTB entry in the LLBP store (i.e. unconditional branches tend to jump further than conditional and their targets are less likely to be cached in the I-cache). Thus, the number of BTB entries per context can be minimised.

3 Methodology

We will evaluate UFP using the ChampSim [9] architectural simulator, comparing its performance against the "ideal" BPU (upper bound), the "bare" FDIP (lower bound), and some prior work specified in Section 4. Our evaluation will focus on measuring execution speed-up, frontend stall reduction, and prefetching accuracy across all three frontend components (branch predictor, BTB, and instruction cache).

We will evaluate across a variety of workloads with large instruction footprints such as web search applications, media streaming applications, online transaction processing, Java traces from benchmark suites such as DaCapo [3], BenchBase[5], Renaissance [24], and Google Traces[6].

4 Related Work

Prior work has proposed various approaches to address the frontend bottleneck for server workloads through prefetching.

Temporal instruction prefetchers such as [7, 8, 11] leverage the observation that programs exhibit repetitive instruction access patterns over time. These prefetchers record and replay instruction cache access sequences. PDIP [10] enhances the traditional FDIP approach by incorporating temporal prefetching of only the targets where FDIP struggles. While these prefetchers reduce I-Cache misses, they focus solely on I-Cache prefetching and require large metadata stores for instruction recording.

Phantom BTB [4] exploits temporal locality in BTB by using a BTB miss as a trigger to replay a sequence of subsequent BTB entries that historically followed the missing entry. In contrast, our context-based mechanism provides a more precise trigger for BTB prefetching by leveraging program control flow information captured in branch contexts, potentially enabling more accurate and timely prefetches.

Twig [13] is a profile-guided BTB prefetching mechanism that identifies critical BTB misses and injects BTB prefetch instructions into the code. Unlike UFP, it prefetches solely to BTB and requires offline profiling to insert the prefetch instructions.

Prior work like Confluence [12], Boomerang [15], and Shotgun [14] has demonstrated the benefits of joint BTB and instruction cache prefetching by recognizing the correlation between BTB and instruction cache misses. However, these approaches do not extend their prefetching mechanisms to include the branch predictor.

Ignite [20] demonstrates the prefetches into the instruction cache, branch predictor, and BTB, but limits its scope to the specific case of serverless function lukewarm starts, leaving the opportunity for a general-purpose unified prefetching solution unexplored.

We plan to quantitatively compare UFP against LLBP, Phantom BTB, Confluence, Boomerang, and Shotgun as these represent the state-of-the-art microarchitectural branch predictor and BTB prefetching solutions for general-purpose workloads.

References

[1] Truls Asheim, Boris Grot, and Rakesh Kumar. 2021. BTB-X: A Storage-Effective BTB Organization. *IEEE Computer Architecture Letters* 20 (2021), 134–137. https://api.semanticscholar.org/CorpusID:238413714

[2] Grant Ayers, Nayana Prasad Nagendra, David I. August, Hyoun Kyu Cho, Svilen Kanev, Christos Kozyrakis, Trivikram Krishnamurthy, Heiner Litz, Tipp Moseley, and Parthasarathy Ranganathan. 2019. AsmDB: understanding and mitigating front-end stalls in warehouse-scale computers. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*. ACM, 462–473.

[3] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, 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. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications* (Portland, OR, USA). ACM Press, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[4] Ioana Burcea and Andreas Moshovos. 2009. Phantom-BTB: a virtualized branch target buffer design. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2009, Washington, DC, USA, March 7-11, 2009*, Mary Lou Soffa and Mary Jane Irwin (Eds.). ACM, 313–324. https://doi.org/10.1145/1508244.1508281

[5] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *PVLDB* 7, 4 (2013), 277–288. http://www.vldb.org/pvldb/vol7/p277-difallah.pdf

[6] DynamoRIO. 2024. *Google Workload Traces*. Retrieved April 01, 2024 from https://dynamorio.org/google_workload_traces.html

[7] Michael Ferdman, Cansu Kaynak, and Babak Falsafi. 2011. Proactive instruction fetch. *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2011), 152–162. https://api.semanticscholar.org/CorpusID:9228401

[8] Michael Ferdman, Thomas F. Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2008. Temporal instruction fetch streaming. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 1–10.

[9] Nathan Gober, Gino Chacon, Lei Wang, Paul V. Gratz, Daniel A. Jimenez, Elvira Teran, Seth Pugsley, and Jinchun Kim. 2022. The Championship Simulator: Architectural Simulation for Education and Competition. (2022). https://doi.org/10.48550/arXiv.2210.14324

[10] Bhargav Reddy Godala, Sankara Prasad Ramesh, Gilles A. Pokam, Jared Stark, André Seznec, Dean M. Tullsen, and David I. August. 2024. PDIP: Priority Directed Instruction Prefetching. *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (2024). https://api.semanticscholar.org/CorpusID:267053631

[11] Cansu Kaynak, Boris Grot, and Babak Falsafi. 2013. SHIFT: Shared history instruction fetch for lean-core server processors. *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2013), 272–283. https://api.semanticscholar.org/CorpusID:8077124

[12] Cansu Kaynak, Boris Grot, and Babak Falsafi. 2015. Confluence: Unified instruction supply for scale-out servers. *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2015), 166–177. https://api.semanticscholar.org/CorpusID:16525664

[13] Tanvir Ahmed Khan, Nathan Brown, Akshitha Sriraman, Niranjan K. Soundararajan, Rakesh Kumar, Joseph Devietti, Sreenivas Subramoney, Gilles A. Pokam, Heiner Litz, and Baris Kasikci. 2021. Twig: Profile-Guided BTB Prefetching for Data Center Applications. *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (2021). https://api.semanticscholar.org/CorpusID:237512130

[14] Rakesh Kumar, Boris Grot, and Vijay Nagarajan. 2018. Blasting through the Front-End Bottleneck with Shotgun. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIII)*. ACM, 30–42.

[15] Rakesh Kumar, Cheng-Chieh Huang, Boris Grot, and Vijay Nagarajan. 2017. Boomerang: A Metadata-Free Architecture for Control Flow Delivery. In *Proceedings of the 23rd IEEE Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, 493–504.

[16] Yunzhe Liu, Xinyu Li, Tingting Zhang, Tianyi Liu, Qi Guo, Fuxin Zhang, and Jian Wang. 2024. AVM-BTB: Adaptive and Virtualized Multi-level Branch Target Buffer. *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)* (2024), 17–31. https://api.semanticscholar.org/CorpusID:271646613

[17] Pierre Michaud. 2005. A PPM-like, Tag-based Predictor. *J. Instr. Level Parallelism* 7 (2005).

[18] Glenn Reinman, Brad Calder, and Todd M. Austin. 1999. Fetch Directed Instruction Prefetching. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM/IEEE Computer Society, 16–27.

[19] Glenn D. Reinman, Todd M. Austin, and Brad Calder. 1999. A scalable front-end architecture for fast instruction delivery. *Proceedings of the 26th International Symposium on Computer Architecture (Cat. No.99CB36367)* (1999), 234–245. https://api.semanticscholar.org/CorpusID:713202

[20] David Schall, Andreas Sandberg, and Boris Grot. 2023. Warming Up a Cold Front-End with Ignite. *2023 56th IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2023), 254–267. https://api.semanticscholar.org/CorpusID:263743386

[21] David Schall, Andreas Sandberg, and Boris Grot. 2024. The Last-Level Branch Predictor. In *Proceedings of the 57th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '24)*. IEEE.

[22] André Seznec. 2016. TAGE-SC-L Branch Predictors Again. In *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*.

[23] Niranjan K. Soundararajan, Peter Braun, Tanvir Ahmed Khan, Baris Kasikci, Heiner Litz, and Sreenivas Subramoney. 2021. PDede: Partitioned, Deduplicated, Delta Branch Target Buffer. *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (2021). https://api.semanticscholar.org/CorpusID:237511563

[24] Renaissance Suite. 2024. *Renaissance Suite: A modern benchmark suite for the JVM*. Retrieved April 01, 2024 from https://renaissance.dev/