Rescuing Data Center Processors

by

Tanvir Ahmed Khan

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2023

Doctoral Committee:

Assistant Professor Baris Kasikci, Chair
Professor Todd M. Austin
Assistant Professor Jean-Baptiste Jeannin
Professor Shan Lu, University of Chicago
Professor Scott A. Mahlke

Tanvir Ahmed Khan
takh@umich.edu
ORCID iD: 0000-0003-3741-1455

© Tanvir Ahmed Khan 2023

ACKNOWLEDGMENTS

I would like to thank my advisor Baris Kasikci to let me be an independent researcher. I would also like to thank Todd M. Austin, Jean-Baptiste Jeannin, Shan Lu, and Scott A. Mahlke for being in my dissertation committee.

I would also like to thank undergraduate students who worked with me for the last few years: Yifan Zhao, Xiaohe Cheng, Shariq Hafeez, Yineng Yan, Ashfaqur Rahaman, Dexin Zhang, Nathan Brown, Scott Hadley, Zhenhang He, Muhammed Ugur, Shixin Song, Diane Chiang, and Kan Zhu.

I would also like to thank industry researchers who have helped me with insightful feedback over the years: Gagan Gupta, Rathijit Sen, Gilles Pokam, Maksim Panchenko, Guilherme Ottoni, Niranjan Soundararajan, Sreenivas Subramoney, Krishnendra Nathella, Dam Sunwoo, Jumana Mundichipparakkal, Jaekyu Lee, Matt Horsnell, Akanksha Jain, Derek Bruening, Abhinav Sharma, Sangeeta Bhattacharya, and Victor Lee.

I would also like to thank my fellow PhD students without whom I could not have survived graduate school: Jiacheng Ma, Ian Neal, Kevin Loughlin, Andrew Loveless, Marina Minkin, Andrew Quinn, Akshitha Sriraman, and Gefei Zuo.

I would also like to thank Yuhan Chen, Yuxuan Zhang, Peter Braun, Sara Mahdizadeh Shahri, Truls Asheim, Surim Oh, Saba Jamilan, Pooneh Safayenikoo, Yiwei Yang, Scott Beamer, Ali Ansari, Shanqing Lin, Fangjia Shen, Siavash Zangeneh, Jennifer B. Sartor, Michael Stumm, Babak Falsafi, Daniel A. Jiménez, Joseph Devietti, Barzan Mozafari, Trevor Mudge, Rakesh Kumar, Cheng Jiang, Simon Peter, Nishil Talati, Jingyuan Zhu, Alireza Khadem, Xin He, Youngjin Kwon, Eric Shiple, Alex Erf, Brian Noble, Westley Weimer, Thomas Wenisch, Nikola Banovic, Valeria Bertacco, Mosharaf Chowdhury, Manos Kapritsos, Mohammed Islam, Emily Mower Provost, Cyrus Omar, Xinyu Wang, Atul Prakash, Satish Narayanasamy, Reetuparna Das, Roya Ensafi, Max New, Daniel Genkin, Paul Grubbs, Justin Johnson, Jason Mars, Alanson Sample, Kang G. Shin, Yatin Manerkar, George Tzimpragos, Hasan Al Maruf, Mouaz Chowdhury, Mert D. Pesé, Chun-Yu (Daniel) Chen, Ram Srivatsa Kannan, Eli Goldweber, Juncheng Gu, Dongyao Chen, Tony (Nuda) Zhang, Arun Ganesan, Remzi Can Aksoy, Youngmoon Lee, Hossein Golestani, Steve Zekeny, Amir Mirhosseini, Subarno Banerjee, Md Salman Nazir, Anindya Das Antar, Kishwar Mashooq, Soumitra Roy Joy, Jamie Goldsmith, Stephen Reger, Zachary Champion, Stephanie Jones, Magdalena Calvillo, Laura Fink, George Kroslak, Karen Liska, and Heiner Litz.

My work was supported by Rackham Predoctoral Fellowship, Rollin M. Gerstacker Foundation Fellowship, Intel and Google gifts, Semiconductor Research Corporation (SRC) seed and realignment grants, NSF/Intel Partnership on Foundational Microarchitecture Research (FoMR) grants, and the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

Finally, I am truly grateful to my amazing parents, Rowsan Ara Begum and Md. Abdul Alim, for their continuous sacrifices throughout my life. I would also like to thank my family and friends for their unwavering support. And last but not least, a special shoutout to my incredible wife, Farjia Rashid, for standing by my side throughout my PhD journey. I could not have done it without her!

TABLE OF CONTENTS

ACKNOWLEDGMENTS ii

LIST OF FIGURES viii

LIST OF TABLES xv

ABSTRACT xvi

CHAPTER

1 Introduction 1

1.1 Problem Overview 1
1.2 Solution Overview 3
1.3 Summary of Contributions 5
1.4 Outline and Previously Published Work 6

2 DMon: Efficient Detection and Correction of Data Locality Problems Using Selective Profiling 7

2.1 Introduction 7
2.2 Challenges 9
2.3 Selective Profiling 10
2.3.1 Targeted Monitoring 11
2.3.2 Incremental Monitoring 12
2.3.3 Sampling 14
2.4 DMon 14
2.4.1 Static Memory Access Pattern Analysis 15
2.4.2 Optimizations Implemented in DMon 18
2.5 Implementation 22
2.6 Evaluation 22
2.6.1 Selective Profiling Efficiency 23
2.6.2 Effectiveness 24
2.6.3 Real-World Case Studies 29
2.6.4 Sensitivity Analysis 31
2.7 Related Work 34
2.8 Conclusion 35

3 Huron: Hybrid False Sharing Detection and Repair 36

3.1 Introduction . . . 36
3.2 Background and Challenges . . . 38
3.2.1 Effectiveness . . . 39
3.2.2 Efficiency . . . 39
3.2.3 Accuracy . . . 40
3.3 Design . . . 41
3.3.1 Instrumentation Pass . . . 41
3.3.2 In-House False Sharing Detection . . . 42
3.3.3 Memory Layout Transformation Pass . . . 45
3.3.4 Input-Independent False Sharing Repair . . . 48
3.3.5 In-Production False Sharing Detection and Repair . . . 50
3.4 Implementation . . . 51
3.5 Evaluation . . . 51
3.5.1 Experimental Setup . . . 51
3.5.2 Accuracy of False Sharing Detection . . . 53
3.5.3 Ability to Repair False Sharing Bugs . . . 55
3.5.4 Effectiveness Comparison to State of the Art . . . 55
3.5.5 Effectiveness of Huron's Input-Independent Repair . . . 56
3.5.6 Impact of Test Cases on Effectiveness . . . 57
3.5.7 False Sharing Repair Overhead . . . 57
3.5.8 Overhead of Huron's In-House Detection . . . 59
3.5.9 Effect of False Sharing Cache on In-Production Overhead . . . 60
3.5.10 Contributions of In-House and In-Production Repair Techniques . . . 60
3.5.11 Effect of Detection Time Window Granularity on Repair Speedup . . . 61
3.6 Related Work . . . 62
3.7 Conclusion . . . 63

4 I-SPY: Context-Driven Conditional Instruction Prefetching with Coalescing . . . 64

4.1 Introduction . . . 65
4.2 Understanding the Challenges of Instruction Prefetching . . . 67
4.2.1 What Information is Needed to Efficiently Predict an I-Cache Miss? . . . 68
4.2.2 When To Prefetch an Instruction? . . . 69
4.2.3 Where to Inject a Prefetch? . . . 70
4.2.4 How to Sparingly Prefetch Instructions? . . . 71
4.3 I-SPY . . . 73
4.3.1 Conditional Prefetching . . . 73
4.3.2 Prefetching Coalescing . . . 77
4.4 Usage Model . . . 78
4.5 Evaluation Methodology . . . 80
4.6 Evaluation . . . 82
4.6.1 I-SPY: Performance Analysis . . . 82
4.6.2 I-SPY: Sensitivity Analysis . . . 86
4.7 Discussion . . . 88
4.8 Related Work . . . 91
4.9 Conclusion . . . 92

5 Ripple: Profile-Guided Instruction Cache Replacement for Data Center Applications 93

5.1 Introduction 94
5.2 Why do existing I-cache miss mitigation techniques fall short? 95
5.2.1 Background on evaluated applications 96
5.2.2 Ideal I-cache: The theoretical upper bound 96
5.2.3 Why do modern instruction prefetchers fall short? 97
5.2.4 Why do existing replacement policies fall short? 99
5.3 The Ripple Replacement Mechanism 102
5.3.1 Runtime Profiling 104
5.3.2 Eviction Analysis 104
5.3.3 Injection of Invalidation Instructions 106
5.4 Evaluation 108
5.5 Discussion 115
5.6 Related Work 115
5.7 Conclusion 117

6 Twig: Profile-Guided BTB Prefetching for Data Center Applications 119

6.1 Introduction 119
6.2 Limitations of prior I-Cache & BTB Prefetching Techniques 122
6.2.1 What stops FDIP from eliminating all frontend stalls? 123
6.2.2 Why is a large BTB insufficient for data center applications? 125
6.2.3 Why do existing BTB prefetching mechanisms fall short? 126
6.3 TWIG 131
6.3.1 Software BTB Prefetching 131
6.3.2 BTB Prefetch Coalescing 134
6.4 Evaluation 135
6.4.1 Methodology 135
6.4.2 Performance Analysis 136
6.4.3 Sensitivity Analysis 142
6.5 Related Work 144
6.6 Conclusion 148

7 Whisper: Profile-Guided Branch Misprediction Elimination for Data Center Applications 149

7.1 Introduction 150
7.2 Branch Prediction Challenges for Data Center Applications 152
7.2.1 Experimental methodology 153
7.2.2 Why is branch prediction important for data center applications? 154
7.2.3 Why does the state-of-the-art TAGE-SC-L branch predictor fall short? . . 155
7.2.4 Why do existing profile-guided techniques fall short? 156
7.3 Design of WHISPER 159
7.3.1 Hashed history correlation 161
7.3.2 Randomized formula testing 163
7.3.3 Implication and Converse Non-Implication 164
7.4 Usage Model 166

7.5 Evaluation 169
7.5.1 Methodology 169
7.5.2 Performance analysis 169
7.6 Related Work 175
7.7 Conclusion 177

8 Conclusion and Future work: Democratizing hardware/software co-design 179

BIBLIOGRAPHY 181

LIST OF FIGURES

FIGURE

1.1 Current computer systems hide the speed gap between processor and memory using hardware structures such as instruction cache, branch predictor, and data caches. . . . 2
1.2 The large footprint problem of data center applications 3
1.3 Implications of data center applications' large footprint problem at processor efficiency 4
1.4 Research vision 4
1.5 Research contributions 5

2.1 The locality tree abstraction. Performance events that pertain to each tree node are in *italic*. There are no dedicated events to determine if a program is back-end bound. Instead, selective profiling subtracts from total stalls the sum of the stalls that cause other bottlenecks at layer 1 to determine if an execution is back-end bound. 11
2.2 Incremental monitoring 13
2.3 How DMon leverages selective profiling to detect and repair data locality problems. . 15
2.4 Static memory access pattern analysis in DMon and their corresponding optimizations. Shaded optimizations are mutually exclusive. 17
2.5 Software prefetching for direct memory access, adapted from [261]. The induction variable is of type `int`. The `prefetch` instruction prefetches one cache line (64 bytes). 18
2.6 Software prefetching for indirect memory access, adapted from [27]. 19
2.7 Structure splitting, example adapted from [72]. 20
2.8 Structure merging example. 21
2.9 Monitoring overhead of selective profiling (All $\sigma < 0.02\mu$). 24
2.10 Speedup comparison to AutoFDO (All $\sigma < 0.09\mu$) 26
2.11 Overhead comparison to AutoFDO (All $\sigma < 0.07\mu$) 26
2.12 DMon-generated optimization after observing input #4 generalizes to unseen inputs (All $\sigma < 0.01\mu$). 27
2.13 Input generalization (All $\sigma < 0.04\mu$) 28
2.14 Overhead of DMon-guided optimizations compared to baseline compilation time ($\sigma < 0.1\mu$, log-scaled y). 29
2.15 Speedup due to DMon-guided optimizations for 22 TPC-Hqueries on `PostgreSQL` (All $\sigma < 4.53\%$ of μ). 30
2.16 Speedup provided by selective profile-guided optimizations for just-in-time (JIT) compiled applications against tiered compilation (All $\sigma < 7.68\%$ of μ). 31
2.17 Effect of granularity of in-production time-slice on detection coverage and overhead (All $\sigma < 3.03\%$ of μ). 32

2.18 Effect of incremental monitoring threshold on the coverage of locality problems selective profiling detects across all benchmarks. 33
2.19 Effect of sampling period on the coverage of locality problems selective profiling detects and the average overhead across all benchmarks ($\sigma < 0.01\mu$). 33

3.1 Input-dependent false sharing repair 40
3.2 High-level design of Huron 42
3.3 Listing with false sharing that occurs when multiple threads execute the lines 20 and 21 in `calc_hist`. 43
3.4 Memory layout of `global_t` data structure described in Fig. 3.3 and how the program suffers from false sharing due to thread 1 and thread 2 both incrementing non-overlapping values on the same cache line. 43
3.5 In-house false sharing repair via memory layout transformations for the example in Fig. 3.3. Memory region $[x - y)$ denotes bytes starting from x (inclusive) up to byte y (exclusive). For simplicity, we have omitted access to memory region $[0 - 24)$, *i.e.*, variable `inputs`. Cache line size is 64 bytes. 46
3.6 Speedup comparison to Sheriff [228]. All standard deviations are less than 1.62%. . . 55
3.7 Speedup comparison to TMI [93]. All standard deviations are less than 3.2%. 56
3.8 Speedup comparison to manual repair. All standard deviations are less than 0.71%. . . 56
3.9 Huron' speedup for different input images of `histogram`. Huron generates an input-independent repair for the "small" input. All standard deviations are less than 3.47%. . 57
3.10 Huron's in-house repair that uses the feedback received from its in production component provides greater speedup than TMI. 58
3.11 Memory overhead comparison to Sheriff. The y-axis is log scale. 58
3.12 Memory overhead comparison to TMI. The y-axis is log scale. 59
3.13 Huron achieves up to 41% (15% on average) higher speedup than Sheriff when we limit the memory for the `lu_ncb` benchmark. All standard deviations are less than 0.59%. 59
3.14 Huron achieves up to 214% (49% average) higher speedup than TMI when we limit the memory for the `lu_ncb` benchmark. All standard deviations are less than 4.16%. . 59
3.15 Overhead of Huron's in-house detection, in seconds. 60
3.16 Speedup of in-production false sharing detection using Huron's false sharing cache. . . 60
3.17 The effect of detection window granularity on Huron's speedup (`lockless_writer` benchmark). 60
3.18 Huron's provides greater performance speedup as more false sharing instances are repaired in house. All standard deviations are less than 3.62%. 61

4.1 Several widely-used data center applications spend a significant fraction of their pipeline slots on "Frontend-bound" stalls, waiting for I-cache misses to return (measured using the Top-down methodology [395]). 67
4.2 A partial example of a miss-annotated dynamic control flow graph. Dashed edges represent execution paths that do not lead to a miss. 69

4.3 Prefetch accuracy vs. miss coverage tradeoff in AsmDB and its relation to ideal cache performance: Miss-coverage increases with an increase in fan-out threshold, but prefetch accuracy starts to reduce. Only 65% of ideal cache performance can be reached at 99% fan-out due to low prefetch accuracy. 71
4.4 AsmDB's static and dynamic code footprint increase: Injecting prefetches in high fan-out predecessors significantly increases static and dynamic code footprints. 72
4.5 Speedup of Contiguous-8 (prefetches all 8 contiguous lines after a miss) vs. Non-contiguous-8 (prefetches only the misses in an 8-line window after a miss): Prefetching non-contiguous cache lines offers a greater speedup opportunity. 73
4.6 An example of I-SPY's context discovery process 75
4.7 Micro-architectural changes needed to execute the context-sensitive conditional prefetch instruction, `Cprefetch` 76
4.8 An example of I-SPY's prefetch coalescing process 77
4.9 Usage model of I-SPY 79
4.10 I-SPY's speedup compared to an ideal cache and AsmDB: I-SPY achieves an average speedup that is 90.4% of ideal. 82
4.11 I-SPY's L1 I-cache MPKI reduction compared with AsmDB: I-SPY removes 15.7% more misses than AsmDB. 83
4.12 Speedup achieved by conditional prefetching and prefetch coalescing over AsmDB: Conditional prefetching often offers better speedup than coalescing, but their combined speedup is significantly better. 84
4.13 I-SPY's prefetch accuracy compared with AsmDB: I-SPY achieves an average of 8.2% better accuracy than AsmDB. 84
4.14 I-SPY's static code footprint increase compared to AsmDB: I-SPY statically injects 37% (average) fewer instructions than AsmDB. 85
4.15 I-SPY's dynamic code footprint increase compared to AsmDB: On average, I-SPY executes 36% fewer prefetch instructions than AsmDB. 86
4.16 I-SPY's performance compared against AsmDB for different application test inputs: I-SPY outperforms AsmDB when the application input differs from the profiled input. 87
4.17 I-SPY's conditional prefetching achieves better performance with an increase in the number of predecessors comprising the context. 88
4.18 I-SPY's average performance variation in response to changes in the minimum (left) and the maximum (right) prefetch distance. 89
4.19 I-SPY's average performance variation in response to increasing the coalescing size: Larger coalescing sizes achieve higher gains. 89
4.20 (left) The probability of coalesced prefetching reduces with an increase in cache line distance. (right) Coalesced prefetch instructions usually bring in less than 4 cache lines. 90
4.21 (left) I-SPY's false positive rate variation in response to an increase in context size: False positives are reduced with a larger context; (right) I-SPY's static code footprint size variation in response to context size: Static code footprint increases with an increase in context size. 90

5.1 Ideal I-cache speedup over an LRU baseline without any prefetching: These data center applications can gain on average 17.7% speedup with an ideal I-cache with no misses. 97

5.2 Fetch directed instruction prefetching (FDIP) speedup over an LRU baseline without any prefetching: FDIP provides 13.4% mean speedup with LRU replacement policy. However, with an ideal cache replacement policy FDIP can provide 16.6% average speedup which is much closer to ideal cache speedup. 98
5.3 Speedup for different cache replacement policies over an LRU baseline with FDIP at the L1 I-cache: None of the existing policies outperform LRU, although an ideal replacement policy provides on average 3.16% speedup. 100
5.4 High-level design of Ripple 103
5.5 An example of Ripple's eviction analysis process 105
5.6 Coverage vs. accuracy trade-off of Ripple for `finagle-http`. Other applications also exhibit a similar trade-off curve. The invalidation threshold providing the best performance across the 9 data center applications we studied varies between 45-65%. . 108
5.7 Ripple's speedup compared to ideal and state-of-the-art replacement policies over an LRU baseline (with different hardware prefetching): On average, Ripple provides 1.6% speedup compared to 3.47% ideal speedup. 111
5.8 Ripple's L1 I-cache miss reduction compared to ideal and state-of-the-art replacement policies over an LRU baseline (with different hardware prefetching): On average, Ripple reduces 19% of all LRU I-cache misses compared to 42.5% miss reduction by the ideal replacement policy. 112
5.9 Ripple's coverage for different applications: On average 50% of replacement requests are processed by evicting cache lines that Ripple invalidates. 113
5.10 Ripple's accuracy for different applications: On average Ripple provides 92% accuracy which ensures that the overall accuracy is 86% even though underlying LRU has an accuracy of 77.8%. 114
5.11 Static instruction overhead introduced by Ripple: On average Ripple inserts 3.4% new static instructions. 115
5.12 Dynamic instruction overhead introduced by Ripple: On average Ripple executes 2.2% extra dynamic instructions. 116
5.13 Ripple's performance for multiple application inputs with the FDIP baseline: On average Ripple provides 17% more speedup with input-specific profiles compared to profiles from different inputs. 117

6.1 Many popular data center applications waste a large portion of their pipeline slots due to "frontend-bound" stalls [147], measured using the Top-down methodology [395]. . 122
6.2 Limit study of FDIP: an ideal I-cache achieves an average 24% speedup, while an ideal BTB provides an average 31% speedup over the FDIP baseline. 123
6.3 BTB Misses Per Kilo Instructions (MPKI) for nine data center applications: these applications experience an average BTB MPKI of 29.7 (8-121). 124
6.4 Breakdown of all BTB misses using 3C miss classification [142]: data center applications suffer BTB misses due to both capacity and conflict issues. 124
6.5 Percentage of capacity misses as BTB size increases from 2K to 64K entries: data center applications require large BTB with at least 32K entries to avoid all capacity misses. For brevity, we show results for only 3 applications, but the behavior is similar across all applications. 125

6.6 Percentage of conflict misses as BTB associativity increases from 4-way to 128-way: data center applications still suffer conflict BTB misses even with an 128-way set-associative BTB. For brevity, we show results for only 3 applications, the behavior is similar across all applications. 126
6.7 Breakdown of all BTB accesses into branch types: conditional branch instructions dominate the total number of BTB accesses 127
6.8 Breakdown of all BTB misses into different branch types: as conditional branch instructions are responsible for most BTB accesses, conditional branch instructions also experience the most number of BTB misses. 127
6.9 Speedups from Shotgun and Confluence over FDIP. 128
6.10 Fraction of BTB misses in temporal streams [367] 129
6.11 Working set size of unconditional branches and calls. Shotgun's U-BTB of 5120 entries is shown in blue. 130
6.12 Percentage of all conditional branches that are outside the range (8 cache lines) of the last executed unconditional branch target. Shotgun cannot prefetch BTB entries for these conditional branches. 130
6.13 An example of how Twig analyzes BTB miss profiles to find accurate and timely prefetch injection site 133
6.14 CDF of branch offset (from the prefetch injection site to the branch instruction) with variation in the number of bits required to store the offset: with just 12-bits Twig stores 80% of all branch offsets for all applications. 134
6.15 CDF of branch target offset with variation in the number of bits required to store the offset: with just 12-bits Twig stores 80% of all branch targets for most applications. . . 135
6.16 Percentage speedup over the FDIP baseline: 32K is for a 32K-entry BTB compared to the 8K-entry baseline BTB. Twig outperforms even the 32K-entry BTB on average with just an 8K-entry BTB with prefetching. 136
6.17 BTB miss coverage of Twig, Confluence, and Shotgun: on average Twig covers 65.4% of all BTB misses. 137
6.18 Contribution of software BTB prefetching and BTB prefetch coalescing toward Twig performance of an ideal BTB: software BTB prefetching provides greater benefits than BTB prefetch coalescing across applications. 138
6.19 Prefetch accuracy of Twig, Confluence, and Shotgun: on average Twig provides 31.3% BTB prefetch accuracy across nine data center applications. 138
6.20 Twig's speedup across different application inputs as the percentage of an ideal BTB performance: Twig trained on a different input provides performance benefits comparable to Twig trained on the same input and outperforms existing BTB prefetching mechanisms. 139
6.21 Static overhead of Twig, measured in % of additional instructions in the binary for a given workload: on average Twig inserts 6% extra static instructions. 141
6.22 Dynamic overhead of Twig, measured in % of additional executed instructions for a given workload: on average Twig incurs only 3% extra dynamic instructions. 141
6.23 % of speedup obtained by Twig compared to an ideal BTB for BTB capacities ranging from 2048 entries to 65536 entries 143
6.24 % of speedup obtained by Twig compared to an ideal BTB for BTB associativity ranging from 4 to 128 143

6.25 Percent of speedup obtained by Twig compared to an ideal BTB for the size of the prefetch buffer, ranging from 8 to 256 144
6.26 Twig's average performance variation in response to increasing the prefetch distance. Across different applications, Twig provides greatest benefit with prefetch distance 15-25 cycles. 145
6.27 Twig's average performance variation in response to changes in the coalesce bitmask size. Twig achieves a majority of the potential performance gains with a 8-bit coalesce bitmask. 145
6.28 % of speedup obtained by Twig compared to an ideal BTB for the size of the FTQ, or maximum distance the decoupled frontend can run ahead, varied between 1 and 64 . . 146

7.1 Data center application limit study: an ideal branch predictor achieves an average IPC speedup of 12.4% (1.3%-26.4%) over the state-of-the-art 64KB TAGE-SC-L baseline. 154
7.2 Branch Mispredictions Per Kilo Instructions (branch-MPKI) for 12 data center applications: 64KB TAGE-SC-L experiences an average branch-MPKI of 3.0 (0.5-7.2) for these applications. 155
7.3 Breakdown of all branch mispredictions among 4 different classes [142, 351, 255]: data center applications suffer from frequent branch mispredictions primarily (76.4% of all mispredictions) due to capacity issues. 156
7.4 Performance of prior profile-guided branch prediction techniques [164, 405] over the 64KB TAGE-SC-L baseline: these techniques reduce only 3.4%-8.9% of all branch mispredictions TAGE-SC-L incurs. Even with unlimited storage, this impractical variant of BranchNet [405] achieves an average misprediction reduction of only 11.9%. . 157
7.5 The distribution of all branch mispredictions across different branch instructions using TAGE-SC-L for SPEC2017 integer benchmarks (left) and data center applications (right). In general, SPEC benchmarks satisfy BranchNet's [405] assumption as only a top-few (*e.g.*, 50) branch instructions cause the majority (*e.g.*, $> 60\%$) of all mispredictions. Data center applications, however, do not satisfy this assumption as mispredictions are distributed across thousands of different branches. 158
7.6 Distributions of all branch mispredictions among different history lengths. Predicting a branch requires correlating its direction with even 1024 prior branch outcomes. . . . 159
7.7 Distributions of branch executions among different logical operations used in the Boolean formula to predict a branch. And (28.9%), always-taken (23.3%), converse non-implication (9.2%), implication (8.8%), never-taken (5.9%), and Or (5.3%) operations together can predict more than 80% of all branch executions. 160
7.8 Micro-architecture of the Read-Once Monotone Boolean Formulas Whisper extends with Implication and Converse Non-Implication. It shows the single unit to predict a branch based on the outcome of the last 2 branches. 165
7.9 Micro-architecture showing how Whisper combines multiple single units in general. This shows how Whisper predicts a branch based on the outcome of the last 8 branches. 166
7.10 Whisper's usage model. 167
7.11 Different components of the `brhint` instruction Whisper proposes. 168

7.12 Speedup over 64KB TAGE-SC-L: Whisper achieves an average speedup of 2.8% (0.4%-4.6%) and outperforms state-of-the-art profile-guided prediction techniques [405, 164]. Whisper's speedup corresponds to 44.1% of the speedup MTAGE-SC offers with unlimited storage [318]. 170
7.13 Whisper's reduction in branch mispredictions compared with BranchNet and ROMBF: Whisper eliminates 7.9% more mispredictions than the best performing realistic prior work. Whisper even removes 4.9% more mispredictions than the unlimited-BranchNet. 171
7.14 Misprediction reduction (%) achieved by hashed history correlation and Implication and Converse Non-Implication over 8-bit ROMBF: hashed history correlation reduces more branch mispredictions than Implication and Converse Non-Implication. 171
7.15 Thanks to randomized formula testing, Whisper achieves high misprediction reduction even after exploring only 0.1% of all formulas (left) while significantly reducing the training time (right, the y-axis is log-10 scale). 172
7.16 Average training time for Whisper compared to prior techniques (the y-axis is log-10 scale): BranchNet requires training times of more than thousands of seconds, even when trained on an NVIDIA Tesla V100 GPU. The training time for ROMBF grows exponentially with an increase in history length. The training time for Whisper is significantly lower than training times for 8-bit ROMBF and BranchNet. 173
7.17 Whisper's performance for various application inputs: On average Whisper reduces 6.6% more branch mispredictions with input-specific profiles compared to profiles from different inputs. 174
7.18 Whisper eliminates more branch mispredictions after merging profiles from various inputs. 174
7.19 Whisper's overhead in static and dynamic instruction increase: on average, Whisper incurs a static overhead of 11.4% (9.8%-13%) and executes 9.8% (5.3%-14.7%) extra dynamic instructions due to `brhint` instructions. 175
7.20 Whisper's reduction in branch mispredictions over the 128KB TAGE-SC-L baseline: Whisper reduces 13.4% of all mispredictions the 128KB TAGE-SC-L incurs. 176
7.21 Whisper's performance for various baseline branch predictor's sizes: Whisper reduces even 1MB TAGE-SC-L's mispredictions by 11.2%. 176
7.22 Whisper's performance for various TAGE-SC-L warm-up periods: Whisper reduces 16.8% of TAGE-SC-L's mispredictions with 50% of instructions considered as warm-up. On the other hand, Whisper avoids 17.5% of TAGE-SC-L's mispredictions without any warm-up. 177
7.23 Whisper's performance for various numbers of simulated instructions: on average, Whisper avoids 14.7% of all mispredictions after simulating one billion instructions. . 178

LIST OF TABLES

TABLE

2.1 Four common memory access patterns that cause data locality problems in many applications. Here, we show their examples from the PARSEC [50] benchmark suite. . . 16
2.2 DMon's detection results of locality problems. 25
2.3 Speedup comparison between DMon and compile-time optimizations. 25

3.1 False sharing detection in existing benchmarks. 54

4.1 Simulated System 81

5.1 Storage overheads of different replacement policies for a 32KB, 8-way set associative instruction cache that has 64B cache lines. 101
5.2 Simulator Parameters 109

6.1 Simulator Parameters 136
6.2 Twig's average speedup across different application inputs with standard deviations. . 140
6.3 Instruction working set size overhead of Twig. 142

7.1 Data center applications and workloads we study. 153
7.2 Simulator parameters 153
7.3 Different design parameters' values. 173

ABSTRACT

Data center applications consume the majority of today's compute cycles. As current computer systems—computer architecture, compilers, and operating systems—are inefficient for data center applications, this dissertation focuses on *redesigning the computer system to enable efficient data center processing*.

The challenges of efficient data center processing are twofold. First, data center applications operate on a large volume of data with complex software functionality to meet the demand of billions of users. Second, processors can no longer provide steady performance scaling to support this rapid growth. This dissertation addresses these challenges by proposing a *feedback loop* in computer systems design.

The *feedback loop* proposed in this dissertation consists of characterization methodologies to find reasons behind the inefficiency and optimization techniques to overcome the inefficiency. This dissertation leverages this feedback loop with profile-guided optimizations that collect data center applications' profiles using characterization methodologies and insert hints utilizing optimization techniques.

While designing this feedback loop, I make two key contributions: (1) I propose systems interfaces using which software can reason about hardware inefficiencies; and (2) I design architectural abstractions using which software can suggest how to avoid hardware inefficiencies. Empowering software to understand and avoid inefficiencies across all major micro-architectural structures, *I make the key contribution of moving the burden of latency-hiding optimizations from hardware to software.*

I help software diagnose hardware problems by designing systems interfaces to characterize hardware inefficiencies faced by data center applications. Drawing insights from diagnosis, my techniques guide software optimizations to avoid hardware inefficiencies. As Moore's Law dwindles, the demand for performance remains ever-present. To satisfy this trending need, data-driven optimizations of existing systems are essential. Systems observability is thus more valuable than ever, but more practically, it is more accessible than ever. Techniques I propose are definitive examples of how systems can proactively use observability to facilitate better communication between hardware and software. Embodying this vision, my systems techniques made proprietary workloads $2\times$ faster. Consequently, I helped companies like ARM adopt my systems interfaces to

diagnose hardware inefficiencies for their data center processors (*e.g.*, ARM Neoverse N1 SDP) that power Amazon Web Service machines, along with Alibaba, and Microsoft data centers.

Hardware optimizations are no longer sufficient for data center applications that process large volumes of data with rapidly growing complex software. Consequently, I design architectural abstractions that move optimizations from hardware to software. Empowering software to avoid inefficiencies across all major micro-architectural structures including instruction cache, data cache, and branch predictor, I redefine the way we design processors. I evaluate all of my techniques for widely-deployed data center applications (*e.g.*, Facebook HHVM, Twitter Finagle, Apache Cassandra, PostgreSQL, MySQL, etc.), and show that they provide significant speedups (more than $2\times$) for these applications. As a result, Intel's data center processors have adopted a couple of my techniques.

Looking forward, I will build open-source systems and benchmarking methodologies to make hardware/software co-design available to a wider audience. I will also use insights from leading these efforts to solve a wide range of efficiency problems across the systems stack.

CHAPTER 1

Introduction

My thesis reexamines the boundary between hardware and software to enable efficient data center processing. I begin this chapter with an overview of the problem: processor inefficiency for data center applications (§1.1). Subsequently, I present an overview of my vision: hardware/software co-design to make processors efficient for data center applications (§1.2). Next, I highlight my key contribution, moving the burden of latency-hiding optimizations from hardware to software. (§1.3). Finally, I conclude with an outline for the rest of this dissertation (§1.4).

1.1 Problem Overview

Data center applications serve billions of people around the world and power nearly every device connected to the Internet. Consequently, millions of processor cores over hundreds of data centers run these applications, incurring large operating costs and carbon emissions. As data center applications are extremely inefficient, the goal of this dissertation's research is to enable efficient data center processing. Let me first motivate why we need to enable efficient data center processing with the example of *Google Web Search*.

Google Web Search: A Motivating Example.

The typical latency of a Google web search is around hundreds of milliseconds [59]. To serve a single search query, Google have to use 8 processor cores [39]. Moreover, during those milliseconds, the efficiency for these cores are only around 32% [41].

Now, consider serving billions of users all over the planet. Serving so many users means companies like Google and Meta deploy millions of processor cores in their data centers [124]. Deploying so many processors over hundreds of data centers incur millions of dollars in operating expenses [344]. More importantly, these processors are responsible for major energy cost [22]. For example, even a single percent of efficiency improvement for these data center applications have the same impact as reducing the whole energy usage of a small country like Barbados [248].

In summary, if we can improve the processor efficiency of data center applications to reduce

end-to-end latency, we can save millions of dollars [181, 179], provide better user experience [59], and meaningfully reduce the global energy needs [344]. To do that, let us now understand why processor efficiency is so low for these web services.

Reasons behind Low Processor Efficiency for Data Center Applications.

At its core, a processor takes program instructions, performs the corresponding operations, and stores the data outcome of the operation. Ideally, the processor should complete such a full pipelined operation on each cycle. However, for Google web-search, processors were performing useful operations only on 32% of the time, wasting 68% of the remaining time and energy [39, 41, 40]. To understand this low efficiency, we will have to look at how processors access these instructions and data. Processors access both instructions and data from memory, and there is a significant ($\approx 100\times$) speed gap between processors and memory [157].

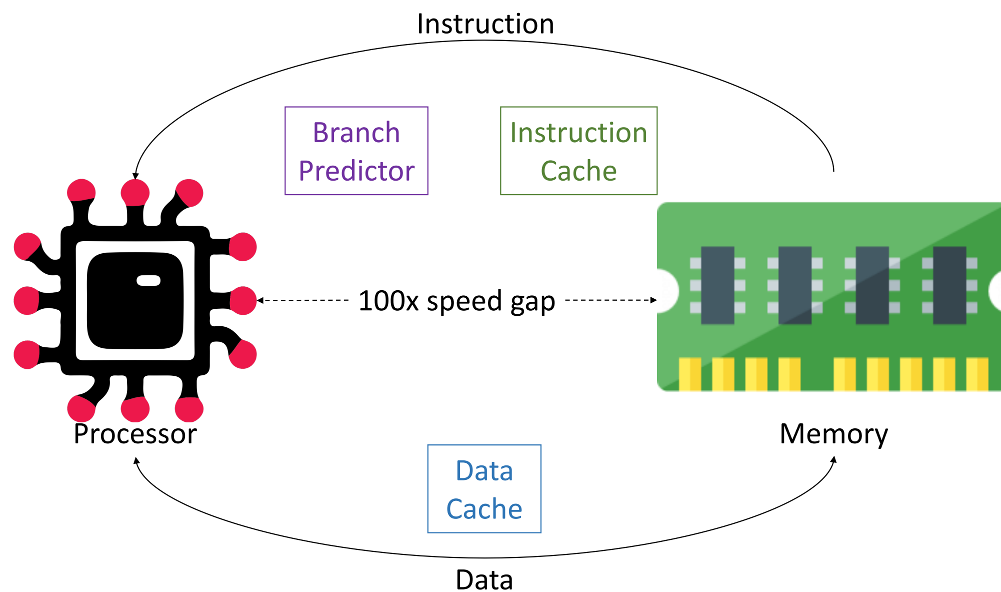


Figure 1.1: Current computer systems hide the speed gap between processor and memory using hardware structures such as instruction cache, branch predictor, and data caches.

To hide this speed gap, as shown in Fig. 1.1, current computer systems primarily rely on hardware structures. Hardware structures such as instruction cache and branch predictor on the instruction side, and data caches on the data side. On the instruction side, instruction cache contains frequently and recently executed instructions and branch predictor contains prior directions of control flow or branch instructions to predict next instructions to execute.

On the data side, processors use different data caches that again contain frequently and recently executed data items. In particular, there is a hierarchy of caches such as L1, L2, and L3 cache. L1 is smallest and fastest while L3 is the largest but slowest. Unfortunately, these structures—instruction cache, branch predictor, and data caches—are no longer sufficient for data center applications. Let

me explain this insufficiency with the example of instruction cache, as shown in Fig. 1.2.

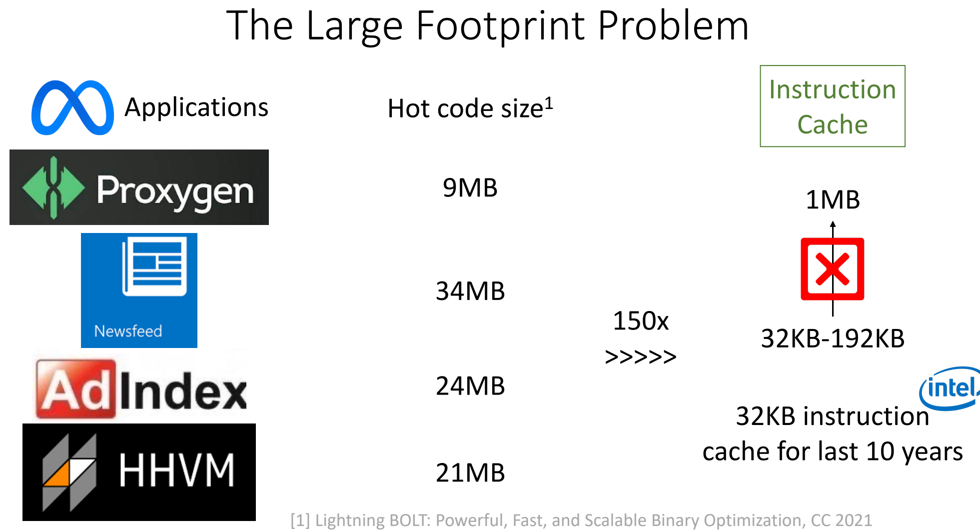


Figure 1.2: The large footprint problem of data center applications

In state-of-the-art processors, instruction cache sizes are around hundreds of KiloBytes (KBs). However, if we look at data center applications, say popular Meta applications, we see that their hot or frequently executed code sizes are around tens of MegaBytes [279]. Therefore, there is a significant ($\approx 150\times$) gap between application footprints and instruction cache sizes. Furthermore, as hardware performance scaling comes to an end, we can no longer increase the instruction cache sizes exponentially [338]. For example, instruction caches in various generations of Intel processors had the same size of 32 KBs for the last 10 years [407].

Now, let's see the implications of this large footprint problem at processor efficiency, as I show in Fig. 1.3. For example, hypothetically, if we had an instruction cache of 10s of megabytes, the processor efficiency for the Google Web Search would get a boost of 24% [39]. Similarly, if we had a branch predictor of several megabytes, the processor efficiency for the Google Web Search would get a boost of 15% [41]. Finally, if we had a data cache of tens of gigabytes, the processor efficiency for the Google Web Search would get a boost of 21% [40]. Unfortunately, hardware technology can no longer provide such an exponential increase [337].

1.2 Solution Overview

As hardware-only solutions are no longer sufficient for data center applications, **my key vision is to redesign computer systems so that hardware and software can work together to make processors efficient.** However, it is really challenging to realize this vision. In many cases, software

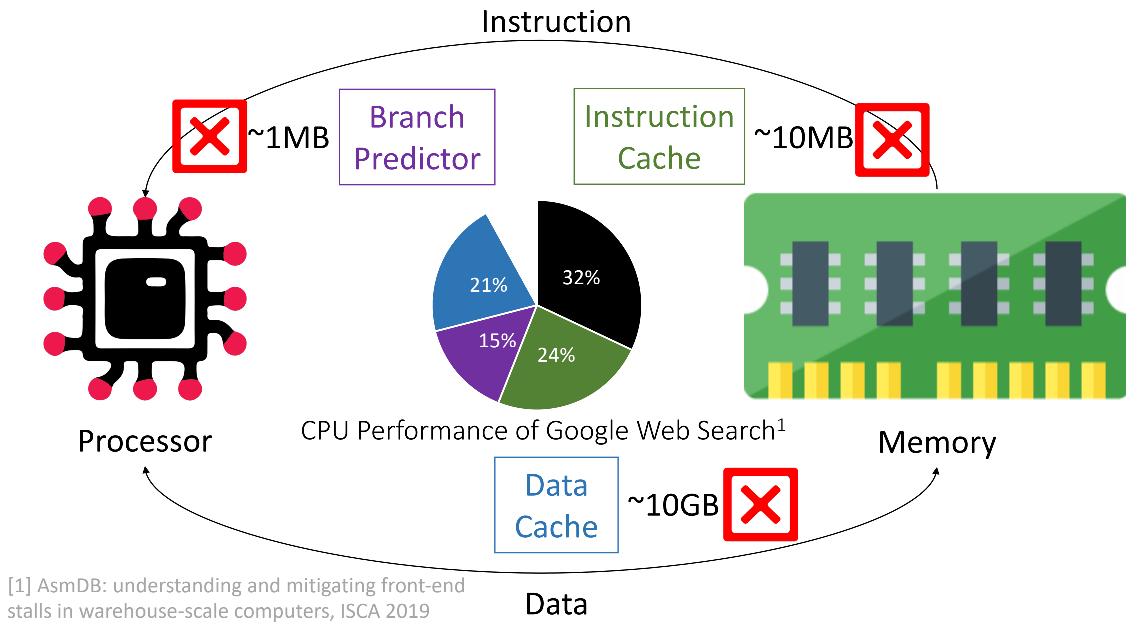


Figure 1.3: Implications of data center applications' large footprint problem at processor efficiency

does not have the capability to diagnose hardware problems. Moreover, in most cases, software does not have the capability to solve hardware problems.

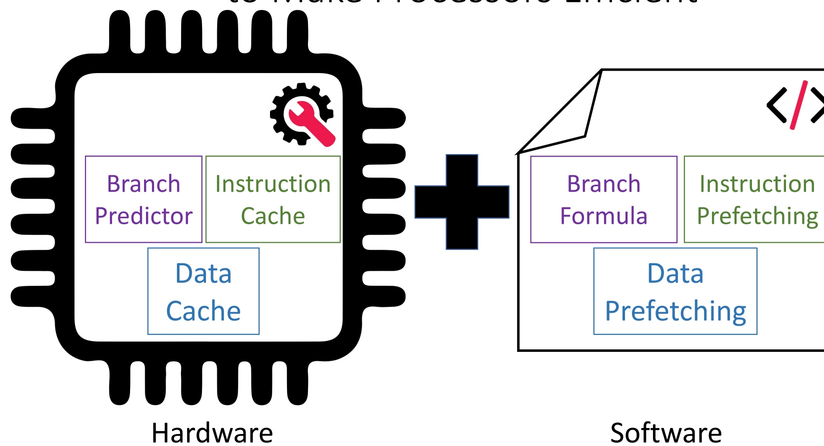


Figure 1.4: Research vision

I address these challenges using three concrete steps. First, I propose characterization methodologies to help software diagnose hardware inefficiencies. In particular, these methodologies extracts meaningful insights by carefully analyzing hardware performance counter events. Second, I design optimization techniques to help software avoid hardware inefficiencies. Specifically, these techniques draw insights from computer architecture, operating systems, and compilers. Finally, I

evaluate my proposed techniques on real-world data center applications used by companies such as Meta, Twitter, and Netflix to validate their effectiveness.

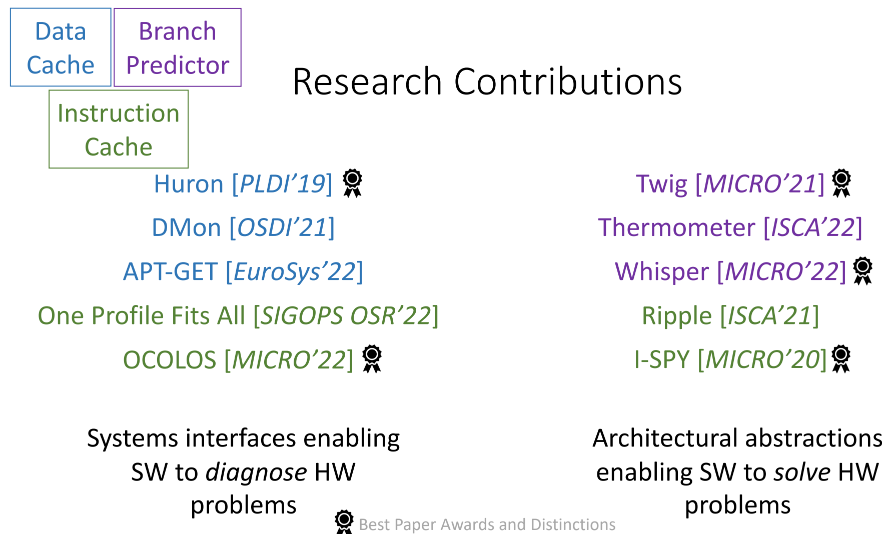


Figure 1.5: Research contributions

1.3 Summary of Contributions

My dissertation's key contribution is moving the burden of latency-hiding optimizations from hardware to software. My contribution has two primary components: (1) I design systems interfaces using which software can reason about hardware inefficiencies, and (2) I propose architectural abstractions using which software can suggest how to avoid hardware inefficiencies. By empowering software to understand and avoid inefficiencies across all major micro-architectural structures, including data cache, instruction cache, and branch predictor (as depicted in Fig. 1.5), I redefine the way we design processors. Consequently, my work presented in this dissertation has been recognized with several Best Paper Awards and Distinctions.

Systems Interfaces to Help Software Diagnose Hardware Problems.

The first contribution of this dissertation is the design of several systems interfaces [203, 196, 270, 361, 157] that enable software to diagnose hardware problems. I demonstrate this contribution with the example of DMon [196].

Performance inefficiency of data center applications can only be observed in production [407]. Therefore, the key challenge of observing hardware inefficiencies and their root causes is to perform the observation with a low-enough overhead that is suitable for production use. DMon addresses this challenge by gathering run-time information selectively and incrementally. DMon continuously monitors production executions only for symptoms of hardware inefficiencies (*e.g.*,

frequent memory stalls and increased cache misses). When DMon observes these high-level indicators of hardware inefficiencies, it automatically transitions to incrementally monitoring more precise information about the exact cause of the hardware inefficiency. To perform the transition, DMon proposes a hierarchical abstraction of hardware events to create an accurate understanding of the underlying inefficiency with only 1.4% overhead. Using insights from the understanding, DMon guides optimizations to eliminate inefficiencies. As a result, DMon speeds up PostgreSQL, a popular database systems, by 17%.

Architectural Abstractions to Help Software Avoid Hardware Problems.

The second contribution of this dissertation is the design of several architectural abstractions [198, 201, 194, 333, 200] that enable software to solve hardware problems. I demonstrate this contribution with the example of Whisper [200].

Whisper investigates branch (*i.e.*, control flow) instructions' behavior in data center applications to show that large code footprints of these applications trigger frequent branch mispredictions and subsequent instruction fetch bottlenecks, wasting 21% of all CPU cycles. Specifically, Whisper's investigation reveals that hardware techniques cannot avoid mispredictions even with an infeasible amount (tens of megabytes) of on-chip storage, also explaining why research into improving branch prediction accuracy has been at an impasse for over a decade.

Whisper solves the capacity problem by moving the prediction of hard-to-predict branches from hardware to software. Whisper leverages data center applications' production profiles to identify hard-to-predict branches. To predict these branches accurately, Whisper finds correlations between their directions and many prior branch directions. Whisper efficiently encodes these correlations in software using Boolean formulas and improves the overall efficacy of branch prediction. Consequently, Whisper received the MICRO'22 Best Paper Award.

1.4 Outline and Previously Published Work

The rest of this dissertation proceeds as follows: Chapters 2 and 3 describe how to enable software to accurately and efficiently diagnose hardware problems, with the example of data cache. Chapters 4, 5, 6, and 7 demonstrate how to empower software to effectively solve hardware problems, with examples of instruction cache and branch predictor. Chapter 8 concludes by describing a roadmap of future work to realize the vision of ubiquitous hardware/software co-design.

Chapter 2 revises material from [196]. Chapter 3 incorporates content from [203]. Chapter 4 builds upon the work presented in [198]. Chapter 5 revises content from [201]. Chapter 6 incorporates material from [194]. Chapter 7 includes material from [200].

CHAPTER 2

DMon: Efficient Detection and Correction of Data Locality Problems Using Selective Profiling

Poor data locality hurts an application's performance. While compiler-based techniques have been proposed to improve data locality, they depend on heuristics, which can sometimes hurt performance. Therefore, developers typically find data locality issues via dynamic profiling and repair them manually. Alas, existing profiling techniques incur high overhead when used to identify data locality problems and cannot be deployed in production, where programs may exhibit previously-unseen performance problems.

We[1] present selective profiling, a technique that locates data locality problems with low-enough overhead that is suitable for production use. To achieve low overhead, selective profiling gathers runtime execution information selectively and incrementally. Using selective profiling, we build DMon, a system that can automatically locate data locality problems in production, identify access patterns that hurt locality, and repair such patterns using targeted optimizations.

Thanks to selective profiling, DMon's profiling overhead is 1.36% on average, making it feasible for production use. DMon's targeted optimizations provide 16.83% speedup on average (up to 53.14%), compared to a baseline that uses the highest level of compiler optimization. DMon speeds up PostgreSQL, one of the most popular database systems, by 6.64% on average (up to 17.48%).

2.1 Introduction

Poor data locality is the root cause of many performance problems [180, 112, 39]. Rapidly increasing data footprints of modern applications due to heavily data-driven use cases (*e.g.*, analytics [404], machine learning [20], etc.) make matters worse, precipitating data locality problems

[1]Some of the work in this chapter was performed in collaboration with Ian Neal, Gilles Pokam, Barzan Mozafari, and Baris Kasikci [196]. Therefore, I use the "we" pronoun in this chapter to acknowledge their involvement in this work.

further [39]. Recent work shows that up to 64% of all CPU cycles are lost due to poor data locality for widely used data center applications [344].

Although many compiler optimizations aim to eliminate data locality problems statically [261, 260, 27, 73, 72], such optimizations rely on compile-time heuristics, which may not accurately identify and repair problems that manifest dynamically at run time. In fact, as we (§2.6.2) and others [69, 58, 26, 87] demonstrate, compiler-based techniques can sometimes even hurt performance when the assumptions made by those heuristics do not hold in practice.

To overcome the limitations of static optimizations, the systems community has invested substantial effort in developing dynamic profiling tools [370, 379, 127, 220, 88]. Dynamic profilers are capable of gathering detailed and more accurate execution information, which a developer can use to identify and resolve data locality problems.

Traditionally, existing dynamic profiling tools have been used offline, namely during testing and development, where test cases are designed to adequately represent real-world program behavior. However, due to the proliferation of cloud computing and mobile devices, programs exhibit vast variations in terms of how they execute and consume data in production [180, 303]. Consequently, it has become increasingly difficult for offline profiling to be representative of how programs behave in production settings.

Unfortunately, existing dynamic profilers incur considerable overheads when used to detect data locality issues, and therefore they are not suitable for production environments [285, 220, 233, 235, 234, 281, 54].

In this paper, we present *selective profiling*, a data locality profiling technique that not only accurately detects data locality problems, but also incurs low overhead, making it suitable for production deployment. Using selective profiling, we design DMon, a system that can automatically detect and eliminate data locality problems in production systems.

Selective profiling is a lightweight technique to continuously monitor production executions for symptoms of poor data locality (*e.g.*, frequent memory stalls, increased cache misses, etc.). As these high-level indicators of data locality problems are identified, selective profiling automatically transitions to incrementally monitoring more precise information about the source location and exact cause of the data locality problem—this is done by traversing a hierarchical abstraction we introduce, called the *data locality tree* (§2.3), which allows DMon to monitor hardware events in a selective way to create an accurate profile at low run-time overhead.

After gathering the profile, DMon performs an offline analysis to identify common patterns of memory accesses. DMon then matches these patterns to a set of existing data locality optimizations (§2.4.1), which it primarily applies automatically, in a targeted manner (unlike static techniques). For cases where DMon cannot automatically apply an optimization, it provides detailed information about the locality problem to the developer, who can fix the problem manually; in our

evaluation, this case occurs only once and the developer can apply DMon-suggested optimization with minimal effort (¡10 LOC). We provide four optimization passes (§2.4.2) which DMon can use to automatically fix data locality problems and are sufficient for DMon to fix major data locality problems we identify across the systems we test in our evaluation (§2.6).

Selective profiling incurs 1.36% monitoring overhead on average, making it an ideal profiling technique for detecting data locality issues in production. The run-time overhead of selective profiling is significantly (*i.e.*, $9\times$) lower than that of the state-of-the-art data locality profiler [65, 256]. Overall, targeted optimizations performed by DMon for 13 applications deliver on average 16.83% (up to 53.14%) speedup. To show the effectiveness of DMon for large real-world systems, we applied DMon to `PostgreSQL` [348], a popular open-source database system, where DMon-guided optimizations provided on average 6.64% and up to 17.48% speedup across all 22 TPC-H [85] queries. Furthermore, the optimizations enabled by DMon provides 20% more speedup, on average, than optimizations provided by the same state-of-the-art profiler.

Overall, we make the following contributions:

- Selective profiling, a data locality profiling technique that automatically and incrementally monitors fine-grained execution information to accurately detect data locality problems with low overhead.
- DMon, a system that implements selective profiling to detect data locality problems in production systems. DMon automatically selects specific optimizations based on memory access patterns, and applies these well-known optimization techniques automatically in most cases.
- By evaluating DMon in the context of widely-used applications, we show that selective profiling can detect data locality issues in production with low overhead (1.36% on average). Moreover, we show that selective profile-guided targeted data locality optimizations provide significant performance speedup (16.83% on average, up to 53.14%).

We explain the key design challenge for accurately and efficiently detecting data locality problems in §2.2. We describe selective profiling in §2.3, DMon's design in §2.4, and DMon's implementation in §2.5. We evaluate DMon in §2.6, compare DMon to related work in §2.7, and conclude in §2.8.

2.2 Challenges

It is challenging to accurately pinpoint data locality problems, while incurring low run-time performance overhead.

Compiler-based static data locality optimizations [261, 260, 293, 55, 347] are appealing because they incur no run-time overhead. However, static techniques apply optimizations based on compile-time heuristics, which may not accurately identify program locations that suffer from poor

locality at run time. In fact, compiler-based techniques can sometimes even hurt performance when the assumptions made by those heuristics do not hold in practice [69, 58, 26, 87].

To demonstrate how compile-time heuristics can hurt performance, we use a compiler-based data prefetching technique [261] to improve data locality in two matrix decomposition benchmarks [389], `lu_cb` and `lu_ncb` from the PARSEC suite [50]. This optimization combines loop splitting and explicit data prefetching to increase data locality. Using the benchmarks' standard inputs, we determine that 50% of all the cache misses in `lu_cb` and `lu_ncb` stem from a single function, which we optimized using compiler-guided data prefetching [261]. The optimization provides a 19.4% speedup for `lu_ncb`, but yields a 19.85% slowdown for `lu_cb`. This occurs because, for `lu_ncb`, prefetching reduces all cache misses; however, for `lu_cb`, there was a dramatic increase in L2 cache misses despite a reduction in L1 and L3 cache misses.

Dynamic profilers can accurately pinpoint data locality problems [285, 220, 233, 235, 234, 281, 54], however, they impose considerable overhead (*i.e.*, ¿10% on average), as they track too much information: memory accesses, timestamps, cache events, etc. Consequently, existing data locality profilers are not deployed in production.

A potential remedy to the high overhead of existing profilers is statistical sampling, which can collect information with reasonable overhead [42]. For instance, the state-of-the-art Intel VTune profiler [304] samples information such as hardware and software performance counters, timestamps, program locations, and accessed memory addresses to gather the necessary information for detecting data locality issues.

Alas, even sampling is not enough to reduce the overhead incurred by popularly available profilers (*e.g.*, Intel VTune) to detect data locality problems to levels acceptable for production use. To assess the impact of sampling, we use the state-of-the-art profiler VTune to detect the data locality issues in our evaluation targets. Despite sampling-based data collection, VTune still incurs 26% overhead on average (and up to 60%), which is unacceptable for production settings.

We argue that not only the monitored execution information must be deliberately chosen to only pertain to data locality problems, but monitoring must occur incrementally, only when there are increasingly clear signs of poor data locality. Next, we explain how selective profiling achieves this.

2.3 Selective Profiling

Selective profiling is a monitoring technique that incrementally monitors more detailed, yet more targeted, run-time information to identify data locality problems. Next, we discuss the three key components of selective profiling: (1) Targeted Monitoring, (2) Incremental Monitoring, and (3) Sampling.

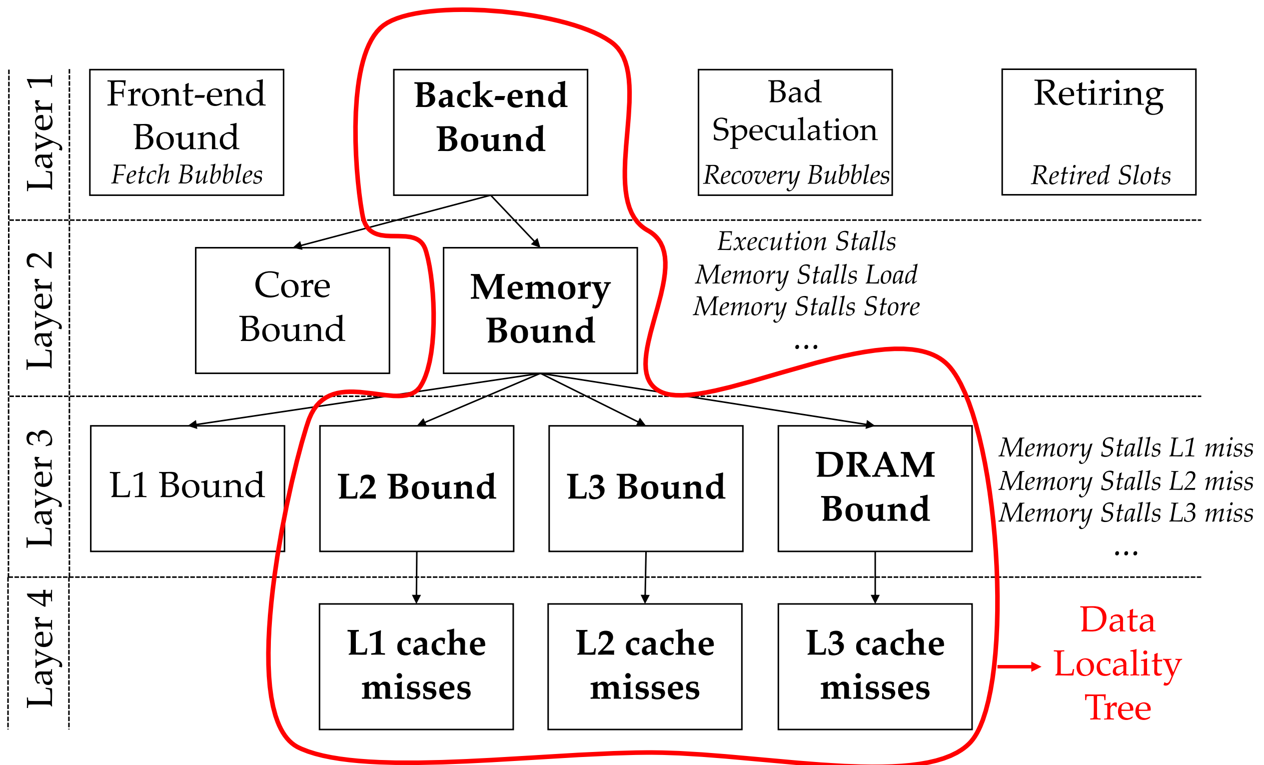


Figure 2.1: The locality tree abstraction. Performance events that pertain to each tree node are in *italic*. There are no dedicated events to determine if a program is back-end bound. Instead, selective profiling subtracts from total stalls the sum of the stalls that cause other bottlenecks at layer 1 to determine if an execution is back-end bound.

2.3.1 Targeted Monitoring

Unlike existing offline profilers [371, 379, 220, 370, 395] that monitor many hardware events and information such as program locations, selective profiling needs to carefully choose which information to monitor in order to accurately and efficiently detect data locality problems. A straw-man approach is to only monitor events such as data cache misses, which are directly related to data locality problems. However, simply monitoring data cache misses in isolation can be misleading. For instance, a seemingly large number of data cache misses may have no impact on the performance of an application that spends a lot of time fetching instructions to execute (a common theme in modern Web services [180, 41]).

Selective profiling monitors a select group of hardware events that allow it to determine if the execution of a program is bounded by a subset of those events that we call the *data locality tree*. As shown in Fig. 2.1, the data locality tree is a hierarchical abstraction of data locality-related performance events from Intel's Top-Down methodology [395]. The Top-Down methodology provides a breakdown of performance events in Intel CPUs, which a developer can use as a guideline to nav-

igate their manual profiling efforts. However, unlike Top-Down, selective profiling automatically transitions from one layer to another, incrementally monitoring more events at each layer of the tree, as increasing evidence of data locality issues is observed at run time.

At layer 1, selective profiling determines whether the execution is back-end bound—*i.e.*, spends a large portion of the time either in CPU execution (CPU bound) or accessing memory (memory bound). At layer 1, a program can also be front-end bound (*i.e.*, fetching instructions), incurring mis-speculations, or retiring instructions. For executions that are back-end bound, selective profiling determines whether they are processor-core bound or memory bound in layer 2.

If an execution is memory bound in layer 2, selective profiling monitors events that provide a breakdown of the execution into 4 categories in layer 3. Of those 4 categories, only 3 are related to data locality problems: L2 bound and L3 bound represent the time spent accessing the L2 and the L3 cache, respectively; "DRAM bound" represents the time spent accessing the DRAM. If a program is L1 bound, the data or instructions that the program uses are already as close to the processor as possible and it is hard to improve data locality further. In such cases, the program may have other performance problems, such as false sharing [360] or lock contention [313].

Selective profiling also tracks information to map performance problems back to code. In layer 4, selective profiling records program location information along with hardware events. For example, if a program is L2 bound, selective profiling records L1 cache misses and the location of the instruction causing the miss. By locating and reducing L1 cache misses, the execution time will potentially not be L2 bound, and the locality problem will likely be fixed. Similarly, if a program is L3 or DRAM bound, selective profiling records L2 and L3 cache misses and associated program locations, respectively.

2.3.2 Incremental Monitoring

Unfortunately, merely restricting the scope of monitored performance events to the data locality tree is not sufficient for low overhead monitoring of data locality issues. Thus, selective profiling instead adopts an incremental monitoring approach. This approach increases the amount of information gathered at run time to efficiently identify program locations that may have a locality problem.

Fig. 2.2 shows the details of incremental monitoring. By default, selective profiling monitors the hardware events that provide the layer 1 breakdown. Selective profiling only transitions to monitoring layer 2 events if the execution is back-end bound for at least 10% of a time-slice p (100ms by default). We use 10% as the default threshold, which we empirically determine to be a reasonable threshold (§2.6.4). We also choose 100ms as a reasonable time-slice for our programs, since the shortest execution across our benchmarks was 1 second and the longest was

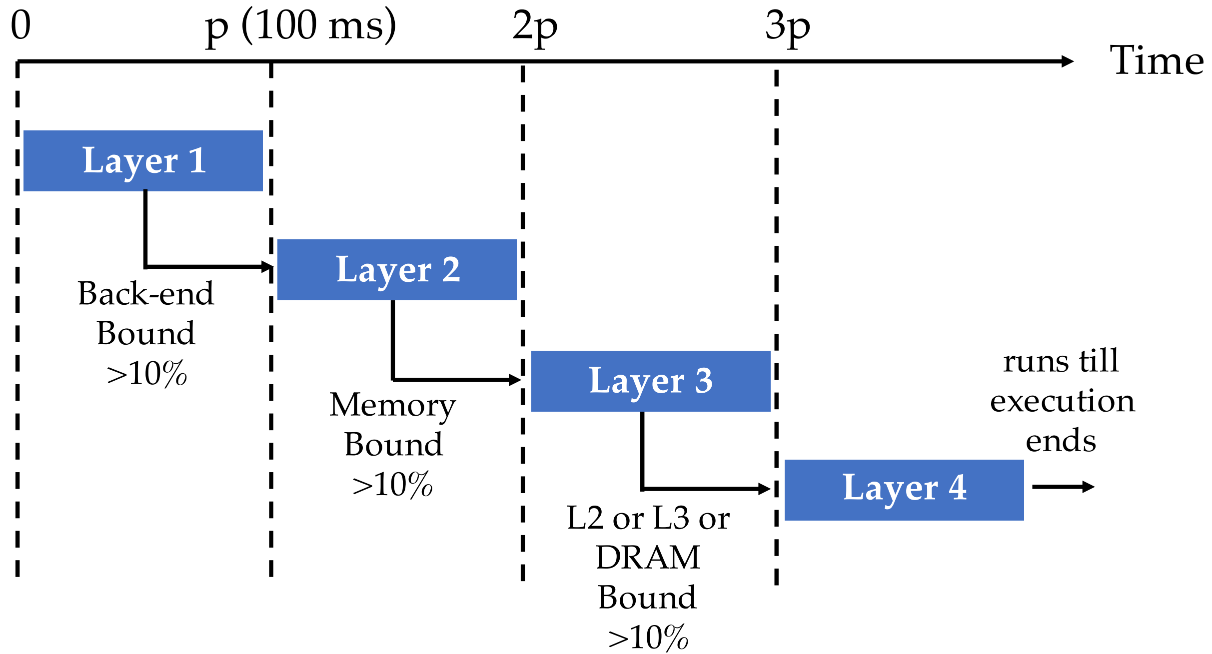


Figure 2.2: Incremental monitoring

2867 seconds. Nonetheless, the percentage and monitoring periods are both configurable. We explore the sensitivity of our results to all these parameters in §2.6.4.

If selective profiling determines that the execution is also memory bound for at least 10% of the same interval p, it starts monitoring layer 3 events. If selective profiling determines that the execution is L2, L3, or DRAM bound for at least 10% of the same interval p, it transitions to layer 4. Selective profiling then gathers L1, L2, and L3 cache miss events and program locations where the misses occur.

Incremental monitoring is key to ensuring selective profiling's low performance overhead. Successive layers are more costly to monitor as they must count more events—for example, layer 2 requires counting $3\times$ more hardware performance events than layer 1. However, unless selective profiling determines that an execution is back-end bound, it only needs to monitor events at layer 1. As shown in §2.6.1, only monitoring layer 1 events incurs a negligible overhead (0.7% on average).

Programs can go through phases of different locality issues (*e.g.*, L2 cache misses in one phase and L3 cache misses in another phase). Selective profiling can pinpoint the root cause of the locality problem for each phase, provided the duration of a given phase is at least $4p$ (where p is the duration of selective profiling's time-slice, per layer). If this time-slice is too long, selective profiling may miss some short-running phases. The time slice is configurable. We empirically determine that a time slice of 100ms is effective in practice (§2.6.4).

2.3.3 Sampling

In addition to targeted and incremental monitoring, selective profiling also employs sampling at layer 4 for recording L1, L2, and L3 cache misses to further reduce the overhead. Although sampling can reduce run-time overhead, it can also reduce the coverage of data locality issues that selective profiling detects if the sampling period is too high. We define coverage as the ratio of the number of locality issues detected with a given sampling rate to the number of locality issues detected with the highest possible sampling rate.

By default, selective profiling uses a conservative sampling period of 1000 (1 sample per 1000 events), which we have empirically found to yield high coverage (97%, discussed in §2.6.4) in detecting locality problems across the 13 benchmarks we evaluated. The developer, however, can use a lower sampling period (up to 1 sample per 100 events, as allowed per Linux's `perf` interface). We analyze the coverage versus overhead trade-off of different sampling periods in §2.6.4.

Selective profiling does not apply sampling in layers 1–3 since sampling reduces coverage. Moreover, in layers 1–3, selective profiling's incremental monitoring reduces the overhead to a negligible amount in all tested applications (on average 1.36%). Therefore, selective profiling does not need to apply sampling at those layers. However, if the overhead of the first three layers is high, selective profiling can optionally enable sampling at those layers as well.

Now, we describe how data locality information collected via selective profiling can be used to guide automated and manual profile-guided optimizations using DMon.

2.4 DMon

Selective profiling detects program locations with poor data locality in production. DMon analyzes these locations offline to identify the data access patterns causing data locality issues. Based on the recognized access patterns, DMon applies existing compiler optimizations only to these program locations in a targeted manner to improve data locality. We offer four such optimizations which we describe in §2.4.2. These optimizations can be automatically applied in most cases for C/C++ applications; for applications written in other programming languages, selective profiling results can still enable manual optimizations (§2.6.3).

Fig. 2.3 shows how DMon employs selective profiling to identify and eliminate data locality issues. In step ①, DMon monitors programs in production to determine whether they suffer from poor locality using selective profiling.

Steps ②–③ happen offline, during recompilation. In step ②, DMon determines the memory access patterns that are causing poor data locality (§2.4.1). In step ③, based on the identified access patterns, either profile-guided automatic optimizations or manual optimizations can be per-

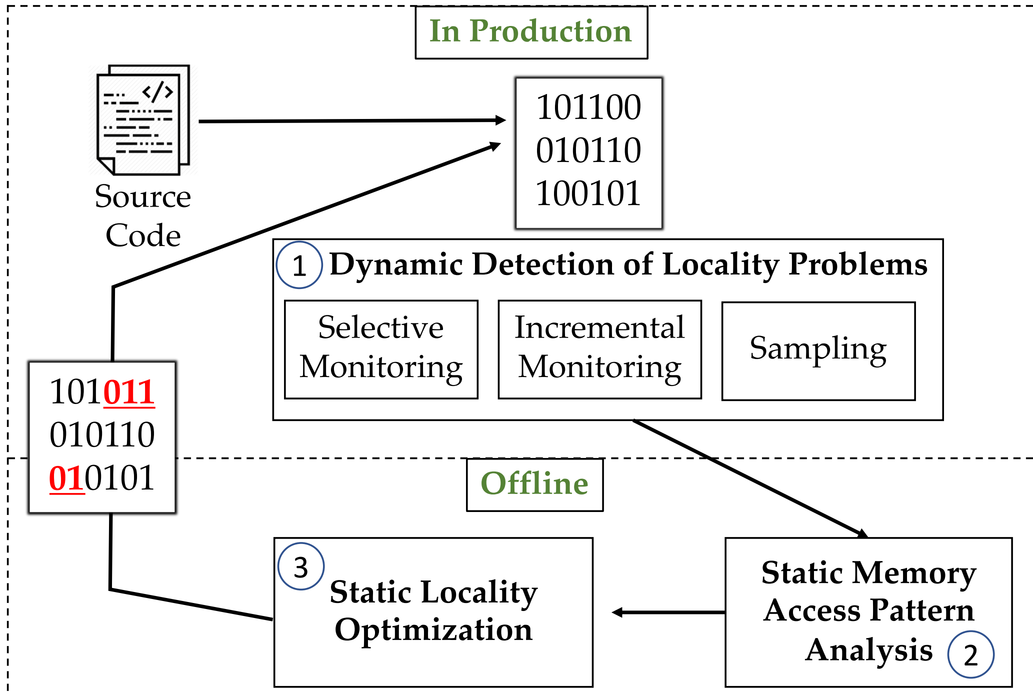


Figure 2.3: How DMon leverages selective profiling to detect and repair data locality problems.

formed to improve data locality (§2.4.2). The optimized program is then rebuilt and redeployed in production.

2.4.1 Static Memory Access Pattern Analysis

Once selective profiling identifies memory access instructions that suffer from poor locality in production, DMon analyzes the corresponding program locations offline to determine the cause of the problems. DMon only analyzes memory access instructions that incur more than 10% of the total cache miss events sampled in layer 4 of selective profiling.

To determine the patterns of data locality issues, we initially analyze the results of selective profiling manually for the benchmarks from the popular PARSEC [50] benchmark suite. Based on our manual analysis of program statements causing data locality issues, we identify four key memory access patterns that can lead to poor data locality. Table 2.1 shows one example of each of these memory access patterns that cause poor data locality. Perhaps unsurprisingly, all the accesses that contribute significantly to poor data locality are in loops that execute many times and access a relatively large amount of data compared to other memory access operations in the application. These four memory access patterns also cause data locality problems in a diverse set of real-world applications (as we show in §2.6.3).

For `lu_ncb`, most cache misses that hurt program performance happen while accessing arrays

Table 2.1: Four common memory access patterns that cause data locality problems in many applications. Here, we show their examples from the PARSEC [50] benchmark suite.

Benchmark	Code snippet	Access pattern
lu_ncb	a[i] += alpha*b[i];	Direct Addressing

radix	this_key = key_from[i] & bb;
	this_key = this_key >> shiftnum;
	tmp = rank_ff_mynum[this_key];
		Indirect Addressing

radiosity	while(int_list)
	{
	  if(int_list->dst==inter->dst)return(1);
	  int_list = int_list->next ;
	}
		Unbalanced Access

dedup	if(LstElmnt->seq.l2num > H->Elmnts[Child]- >seq.l2num){	Pointer Chasing

in a loop. Since the loop induction variable (i) is directly used to index those arrays, we call this pattern *direct addressing*. For radix, the loop induction variable (i) is used to index an auxiliary array to load an intermediate value (this_key). The loaded intermediate value is used as index while accessing another array, and the last access suffers from poor data locality. We categorize this pattern as *indirect addressing*.

For radiosity, most cache misses occur in a while loop, where two member variables (dst and next) of a structure (int_list) are accessed repeatedly. We determine that this structure also contains four other member variables not accessed in this loop. Since only accessing a subset of all member variables causes cache misses, we call this access pattern as *unbalanced access*. Finally, for dedup, locality suffers while accessing a chain of structure pointers (pointers H, Elmnts[Child], and seq, and finally a member variable l2num) in a loop. We denote this pattern as *pointer chasing*.

Based on findings of these manual observations, we design the static memory access pattern analysis component of DMon, as shown in Fig. 2.4. Although DMon's pattern detection is inspired by the manual analysis of locality issues in PARSEC, we show in our evaluation that the patterns DMon identifies generalize to a broad set of systems (§2.6.2 and §2.6.3). In particular, the four patterns of poor locality constitute the root causes of all the data locality problems we discover in nine other benchmarks that we had not studied previously.

As shown in Fig. 2.4, DMon determines the addressing mode of the memory instruction (*i.e.*, direct or indirect addressing). If the access is made to a structure instance, DMon also determines

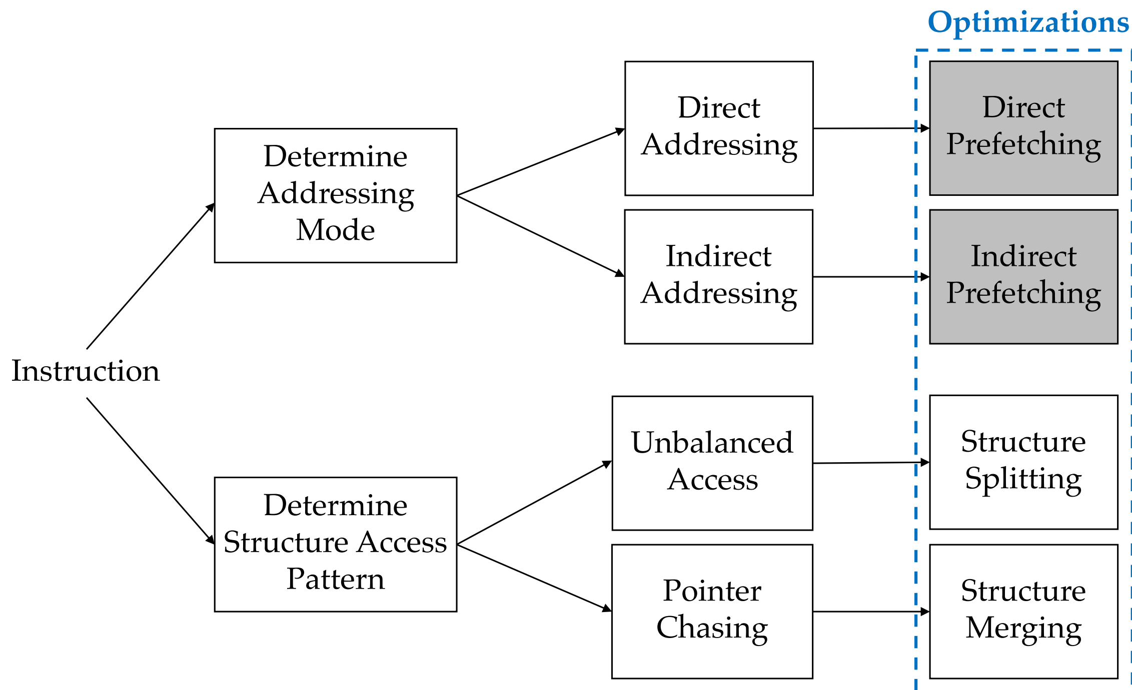


Figure 2.4: Static memory access pattern analysis in DMon and their corresponding optimizations. Shaded optimizations are mutually exclusive.

the type of the access (*i.e.*, unbalanced access and pointer chasing). We discuss each analysis next.

Addressing mode. DMon's static analysis checks if the instruction uses direct or indirect addressing. Here, direct addressing occurs if the computation of the accessed location does not involve another memory address (*e.g.*,`for(i=...) a[i]`). Conversely, indirect memory addressing occurs if the computation of the accessed location involves computing another memory address (*e.g.*,`for(i=...) a[b[i]]`).

Structure access pattern. In addition to determining the addressing mode, DMon's static analysis checks to see if the instruction accesses a member of a structure. DMon does this by mapping the instruction to the compiler intermediate representation and checking if it accesses a structure field. DMon searches for two patterns when a structure member is accessed, namely *unbalanced access* and *pointer chasing*.

DMon concludes that there is an unbalanced access pattern, when accesses to only a *subset* of member variables incur a large fraction of cache misses. Pointer chasing occurs when the accessed memory location belongs to a hierarchy of nested structures (*e.g.*, `A->B->C`).

```
for(i=0;i<128;i++)
  ACCESS a[i];
```

Original Loop

```
for(i=0;i<16;i+=8)
  prefetch(&a[i]);
for(i=0;i<112;i+=8){
  prefetch(&a[i+16]);
  ACCESS a[i], …, a[i+7];
}
for(i=112;i<128;i++)
  ACCESS a[i];
```

Prefetched Loop

Figure 2.5: Software prefetching for direct memory access, adapted from [261]. The induction variable is of type `int`. The `prefetch` instruction prefetches one cache line (64 bytes).

2.4.2 Optimizations Implemented in DMon

To show the usefulness of selective profiling, we implement four profile-guided data locality optimization passes using LLVM [215] for C/C++ programs. Our passes optimize the four patterns of poor data locality that DMon identifies. For applications written in other languages, selective profiling results can be used to apply manual optimizations (§2.6.3).

As shown in Fig. 2.4, DMon recommends applying a specific optimization technique based on the addressing mode and the structure access patterns of the memory access instruction. While these optimizations are well-known and usually applied statically, selective profiling information enables the targeted application of these optimizations to where they are absolutely needed in a program. As we show in §2.6.2, DMon-enabled targeted profile-guided optimizations outperform purely static optimizations by 10% on average.

Direct prefetching. The first optimization we implement uses direct prefetching [261] to fix locality problems that stem from memory accesses that use direct addressing. Direct prefetching fetches the cache lines that a program will access in the near future into the cache to improve data locality.

At a high level, direct prefetching works by splitting each loop suffering from poor data locality into three loops, as shown in Fig. 2.5. The first loop is responsible for prefetching the initial cache line that contains the data accessed by the loop. The second loop starts prefetching the next cache line(s). It also simultaneously performs the original computation that was carried out in the original loop, starting with the first prefetched cache line. The third and last loop completes the computation using the last prefetched cache line.

Direct prefetching can be applied based on compile-time heuristics only. However, this can

```
for(i=0;i<A_SIZE;i++)
  b[a[i]]++;
```

```
for(i=0;i<A_SIZE;i++){
① prefetch(&a[i+16*2]);
  if(i+16<A_SIZE)
②   prefetch(&b[a[i+16]]);
  b[a[i]]++;
}
```

Original Loop   Prefetched Loop

Figure 2.6: Software prefetching for indirect memory access, adapted from [27].

cause significant performance degradation [89], as we also show in our evaluation (§2.6.2). This happens because these heuristics might (1) bloat the code footprint by adding unnecessary prefetching instructions (*e.g.*, for lines that would anyways be prefetched by the hardware prefetcher), and (2) cause cache pollution by prefetching data that is not frequently-accessed.

Direct prefetching can also be applied in hardware with popular hardware prefetchers including next-line and stride prefetchers that most modern processors supposedly employ [364, 141]. However, DMon finds that many directly addressed memory accesses suffer from poor data locality, because the underlying hardware prefetchers can not prefetch the cache lines in a timely manner. This is because prefetchers work in a reactive manner, *i.e.*, it takes several iterations for the hardware prefetcher to detect the pattern and start prefetching, but if prefetching is done with explicit instructions, the performance benefits are immediate.

Instead of applying direct prefetching based on compile-time heuristics, our pass only applies it to program locations where DMon identifies that direct addressing access pattern is causing poor data locality.

Indirect prefetching. Our second optimization uses indirect prefetching [27], which is similar to direct prefetching in that it brings data that will soon be used into the cache. Unlike direct prefetching, indirect prefetching also has to prefetch one additional cache line per each level of indirection.

Fig. 2.6 shows an example of indirect prefetching. Here, the original loop increments elements in an array, `b`. However, the index of the array `b` is computed using another array, `a`. The loop on the right side prefetches the cache line containing the elements of `b` that will be accessed in the near future (prefetch ②). Prefetching the elements of `b` requires accessing the elements of `a`. Thus, to prefetch the elements of `b`, we need to (1) have an array boundary check, and (2) also prefetch the cache line containing the elements of `a` (prefetch ①).

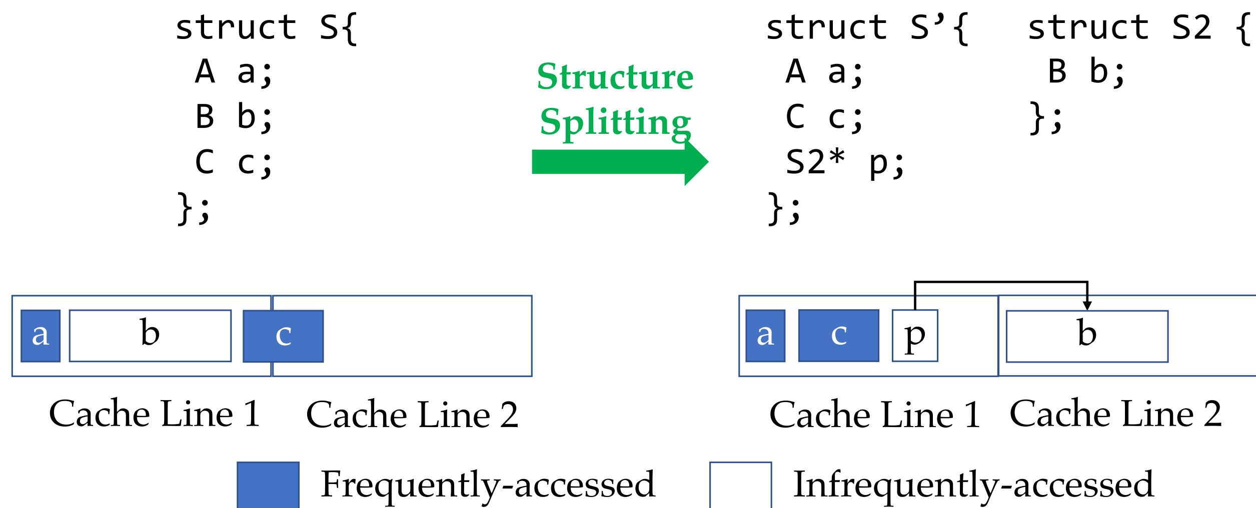


Figure 2.7: Structure splitting, example adapted from [72].

Structure splitting. The third optimization, structure splitting, moves infrequently-accessed members of a structure with a pointer to a new structure that only contains those members. Structure splitting is beneficial only when the total size of infrequently-accessed member(s) is larger than the pointer size. Thus, the size of the original structure is reduced, fitting into fewer cache lines. During memory access pattern analysis, if DMon detects that an unbalanced access pattern (*i.e.*, a subset of structure members are accessed more frequently than others) to members of a structure is causing poor locality, structure splitting is an appropriate optimization.

Fig. 2.7 shows an example of structure splitting. Here, before structure splitting, the structure `S` has three members (`a`, `b`, `c`) of types `A`, `B`, `C`, respectively. In the original program, an instance of `S` spans two cache lines. Both cache lines need to be accessed each time the program accesses an instance of `S`. For example, if neither of these cache lines is present in the L1 cache, the program will incur two L1 cache misses.

After structure splitting, the new structure `S'` fits in a single cache line (Cache Line 1) because the infrequently-accessed member `b` is moved into a new structure `S2`, residing in its own cache line (Cache Line 2). Consequently, when the program accesses an instance of `S`, it will usually only need to access the cache line (Cache Line 1) containing the frequently-accessed members (`a`, `c`), which would incur a single L1 cache miss (rather than two).

Structure splitting has been previously explored [72] in type-safe languages (*e.g.*, Java). However, implementing structure splitting in a type-unsafe language (we target C/C++) is more challenging. This is because structure splitting needs to ensure that the program continues operating correctly when the layout of the structure is modified. More specifically, all the instructions that used to refer to the old layout need to be updated to refer to the new layout.

In our optimization pass, we addresses this challenge using a complete, interprocedural,

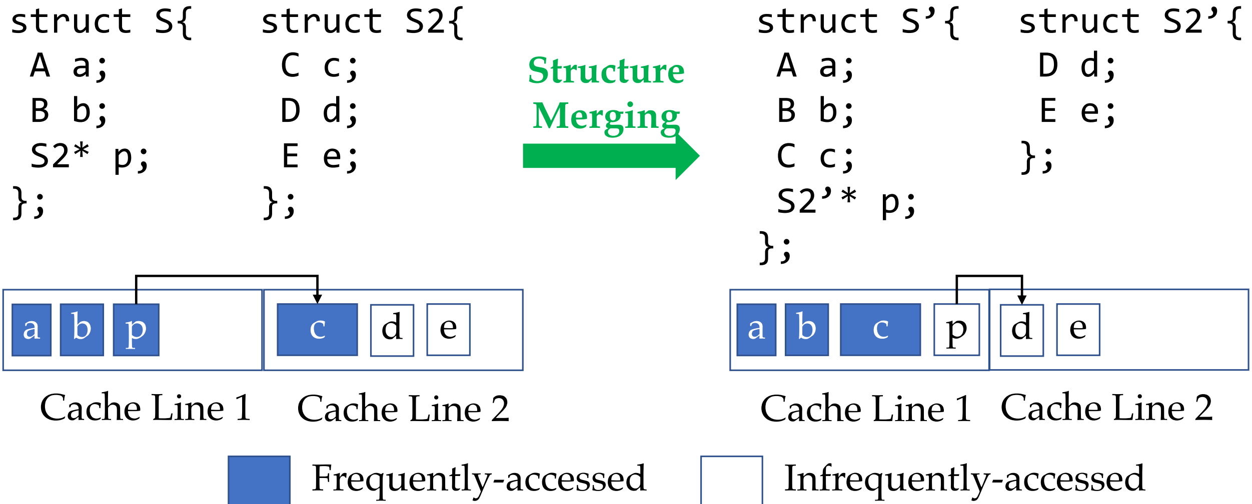


Figure 2.8: Structure merging example.

inclusion-based pointer analysis [30] that can determine all instructions that could possibly access the split structures. As shown in §2.6.2, this optimization can automatically be applied in all but one of the benchmarks.

Structure merging. The final optimization, structure merging, is the inverse of structure splitting as it replaces a frequently-accessed pointer member of a structure with the data that the pointer references. The key idea is to eliminate the pointer chasing pattern that DMon identifies by removing a level of indirection for frequently-accessed elements.

Fig. 2.8 shows an example of structure merging. Before merging, the structure `S` has three members (`a, b, p`) of types `A, B, S2*`, respectively. The instance of `S` resides in the first cache line, and the pointer `p` points to an instance of structure `S2` that resides in the second cache line. The size of `a`, `b`, and `c` is such that they can all fit in one cache line. If `c` is accessed as frequently as `a` and `b`, then data locality can be improved by merging these two structures into one. This structure merge will also bypass one memory access (`S'->C` instead of `S->S2->C`). Structure merging only combines member variables across different structure types and hence does not perform exhaustive data structure conversions (*e.g.*, transforming a linked list into an array) [73, 72].

DMon employs structure merging conservatively so that it will only be applied if soundness can be guaranteed. In other words, DMon applies this optimization only if all updates via the structure pointer can be safely redirected (*e.g.*, in Fig. 2.8, all changes to `S->S2->C` could be replaced by `S'->C`). To ensure this, structure merging also uses the same pointer analysis [30] that structure splitting uses.

Other optimizations.. DMon can be easily extended to accommodate additional optimizations if needed to fix different patterns of memory accesses which cause data locality problems. For example, DMon can work as a framework to apply optimizations like loop reordering, blocking,

tiling, and strip mining in a profile-guided manner. However, many of these optimizations require expensive memory access trace collection which can not be deployed in production due to high overheads [242]. In the future, we intend to explore how these optimizations can be applied based on more efficient profiling.

2.5 Implementation

DMon's selective profiling prototype is implemented for Intel processors. In particular, selective profiling relies on the Linux `perf` [370] interface for profiling hardware events in layers 1–4 (§2.3). We initially build the benchmarks using debug information and the highest level of compiler optimization (`-O3`), and then use the `strip` utility [375] to remove the debug information. During in-production monitoring, selective profiling records the program counter for each sampled cache miss event in layer 4. To efficiently deal with multi-threaded applications, selective profiling maintains a per-thread buffer (2MB per thread) to record the program counters. When the buffer gets full, the previous samples get overwritten. Offline, DMon uses the program counter, the stripped debug information, and the program binary to find the source code location where a cache miss occurred in production.

We implement DMon's optimizations in the LLVM [215] compiler framework. We use `clang` [372] to generate the LLVM intermediate representation (IR) that the optimization passes of DMon can operate on. The optimizations rely on the program's debug information to map the source code location to LLVM IR, because a 1-to-1 mapping between machine code and LLVM IR does not exist.

Similar to other state-of-the-art profile-guided optimization techniques [65, 256], DMon's use of debug information for mapping machine code to LLVM and locating code locations to optimize can introduce inaccuracies. This happens due to optimizations such as inlining. Although it is possible to improve the accuracy of such mapping using more invasive instrumentation and tracing [40], this would be prohibitively costly for production usage [180]. In our evaluation (§2.6), we show that the accuracy provided by debug information can lead to substantial speedup.

The optimizations for structure splitting and structure merging use a whole-program pointer analysis [67].

2.6 Evaluation

In this section, we first evaluate the efficiency of selective profiling by measuring its run-time monitoring overhead. Then, we evaluate the effectiveness of DMon by showing the extent to which fixing the locality problems detected by DMon improves performance of popular benchmarks. Next,

we evaluate selective profiling's generality by applying it to widely-used real-world applications. Finally, we perform sensitivity studies to evaluate how DMon's overhead and detection results vary in response to changes of the different system parameters of DMon.

Software. All experiments are conducted in Ubuntu 18.04 (kernel version 4.15.0-46-generic). The static compiler analyses are implemented in LLVM (7.0.0) on bitcode emitted by clang. Therefore, we use `clang 7` as the baseline compiler.

Hardware. We use a 20-core 2.2 GHz Intel Xeon NUMA (with 2 sockets) machine, with 64 KB of L1-cache (32 KB instruction and 32 KB data), 1024 KB of L2-cache, 14 MB of L3-cache (shared across the same NUMA node), and 96 GB of RAM. Like most Intel processors, each core in the machine uses two hardware prefetchers (next-line and sequential load history driven prefetchers) in the L1 data cache and two hardware prefetchers (adjacent cache line and streaming prefetchers) in the L2 cache [364, 141]. We configure multi-threaded applications and benchmarks to run with 8 threads.

Benchmarks. We use a combination of benchmarks and real-world programs that have been widely used in prior performance profiling and optimization work. In particular, we use all 12 benchmarks from the PARSEC [50] suite, all 11 benchmarks from the SPLASH-2X [388] suite, and all 3 benchmarks written in C from the NPB [43] suite, as well as `HashJoin`, `RandomAccess`, `kcstashtest`, and `DIS`, which are programs with poor data locality from other popular benchmark suites [240, 44, 264, 77]. We also study one of the most popular and heavily-optimized open-source databases, `PostgreSQL` [292], running the TPC-H analytical workload [85]. Finally, we study real-world applications from the Renaissance benchmark suite [295].

Metrics. In all our plots, we report speedup numbers as the ratio between the execution time of the original application compiled with the highest level of optimization (`-O3`) and its run time after applying DMon-guided optimizations. Negative speedup denotes slow-down. Similarly, we report selective profiling overhead as the percentage increase in benchmark execution time while enabling selective profiling. We report performance data as the average of 25 runs in all experiments.

2.6.1 Selective Profiling Efficiency

We evaluate the selective profiling efficiency by studying the overhead selective profiling incurs during dynamic detection of locality problems. Fig. 2.9 shows this overhead. We present results for all the benchmarks we evaluated, including the ones for which selective profiling did not find locality optimization opportunities. For each benchmark, we present the overhead of each layer of monitoring (1–4) that selective profiling employs. Since, selective profiling monitors only one layer at a time, the effective overhead for a given program is less than the maximum overhead across four layers.

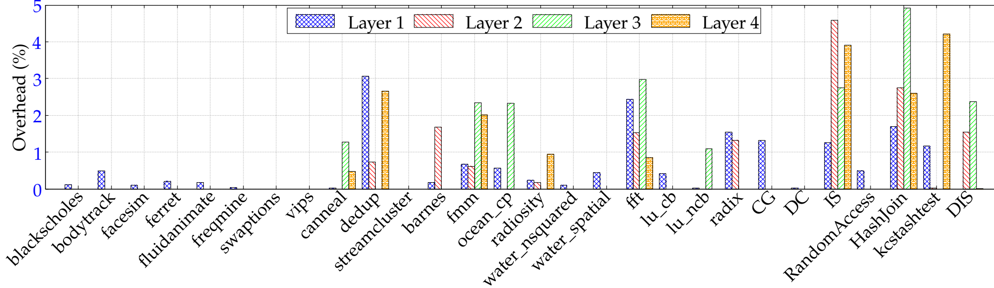


Figure 2.9: Monitoring overhead of selective profiling (All $\sigma < 0.02\mu$).

Across all layers and benchmarks, selective profiling incurs up to 4.92% overhead, and on average only 1.36% overhead. On average, selective profiling incurs an overhead of 0.7% in layer 1, an overhead of 1.5% in layer 2, an overhead of 2.5% in layer 3, and an overhead of 2% in layer 4. For benchmarks that do not have locality problems, layers 2–4 are never triggered.

In only 3 out of all 28 benchmarks, selective profiling incurs more than 3% overhead: `IS` (4.6%), `kcstashtest` (4.2%), and `HashJoin` (4.9%). However, as we detail in §2.6.2, optimizations suggested by DMon also provide greater speedups for these benchmarks than for others (`IS` 30.3%, `kcstashtest` 32.4%, and `HashJoin` 53.1%—compared to 16.83% average speedup enabled by DMon). These benchmarks suffer the most from poor locality, and consequently, selective profiling incurs more overhead to pinpoint the root cause of those problems.

2.6.2 Effectiveness

We evaluate the effectiveness of DMon by studying (1) data locality problems detected by DMon, (2) speedups provided by DMon-guided optimizations, (3) comparison of the speedups provided by DMon-guided optimizations to the speedups provided by Google's AutoFDO [65]—the state-of-the-art profile-guided locality optimization approach, (4) whether DMon-guided optimizations generalize across different program inputs, and (5) the overhead on compilation times due to DMon-guided optimizations.

Locality issues detected by DMon. Table 2.2 summarizes the data locality problems that DMon detects. For brevity, Table 2.2 omits benchmarks where less than 10% of the execution time is bounded by locality problems, as these benchmarks could not benefit from eliminating locality improvements. We also omit these benchmarks in our average performance numbers.

Additionally, Table 2.2 shows the most prominent level of the memory hierarchy for the locality issues detected by selective profiling. Note that, in many cases, DRAM accesses constitute the locality bottlenecks. This is expected, since the highest-latency memory access instructions are served from DRAM. Finally, Table 2.2 also reports the program locations (as "file": "line number")

Table 2.2: DMon's detection results of locality problems.

Benchmark	Execution time (seconds)	Memory hierarchy bottleneck	Program location	Optimization	Automated fix?
canneal	71.8	L3, DRAM	netlist_elem.cpp: 80	Direct Prefetching	Yes
dedup	5.1	DRAM	binheap.c: 93	Structure Merging	Yes
fmm	18.8	DRAM	interactions.C: 169	Structure Splitting	No
				Direct Prefetching	Yes
ocean_cp	36.2	L2, L3, DRAM	multi.C: 273	Direct Prefetching	Yes
radiosity	95.8	L2, L3	rad_tools.C: 399	Structure Splitting	Yes
fft	1.2	DRAM	fft.C: 765	Direct Prefetching	Yes
lu_ncb	47.8	L3, DRAM	lu.C: 466	Direct Prefetching	Yes
radix	6.1	L2, L3, DRAM	radix.C: 624	Indirect Prefetching	Yes
IS	1	L3, DRAM	is.c: 392	Indirect Prefetching	Yes
RandomAccess	607.1	DRAM	randacc.c: 125	Indirect Prefetching	Yes
HashJoin	2867.3	L3, DRAM	npj2epb.c: 300	Indirect Prefetching	Yes
kcstashtest	3.20	L2, L3, DRAM	kcstashdb.h: 146	Direct Prefetching	Yes
DIS	165.3	L2, L3, DRAM	transitive.c: 107	Direct Prefetching	Yes

Table 2.3: Speedup comparison between DMon and compile-time optimizations.

Benchmark	Speedup provided by compile-time optimizations (%)	Speedup provided by DMon (%)
canneal	-7.90	1.07
dedup	-18.90	3.65
fmm	2.83	2.68
ocean_cp	-1.06	2.90
radiosity	-7.14	11.21
fft	1.11	4.57
lu_ncb	3.49	19.40
radix	0.96	1.85
IS	30.52	30.29
RandomAccess	38.83	47.67
HashJoin	9.74	53.14
kcstashtest	37.41	32.39
DIS	-0.28	7.93

that suffer the most from poor locality, along with the optimizations DMon recommends in each case.

As shown, DMon successfully identifies locality problems and suggests appropriate optimizations in each case. In all cases but one (`fmm`), DMon applies optimizations automatically. For `fmm`, while the direct prefetching is applied automatically, structure splitting cannot be applied automatically. This is because, due to excessive type casts, the compile-time optimization cannot exactly determine which program statements may access the modified structure, and therefore cannot automatically update such statements. Nonetheless, since DMon points the developer to the exact source of the locality issue in `fmm`, the fix can easily be applied manually with an 8 LOC update. Moreover, structure splitting and merging can be applied automatically for other applications (`dedup` and `radiosity`) where the automatic transformation can identify and update all statements pointing to the split and merged structures.

Speedup. Table 2.3 compares the speedup provided by the DMon-guided optimizations. Optimizations guided by DMon provide up to 53.14% and on average 16.83% (8% median) speedup. To study the impact of the targeted optimizations guided by selective profiling results, we also report the speedup achieved by the same optimizations if they are applied indiscriminately (*i.e.*, in a non-targeted way), through purely-static compiler passes [261, 27].

As shown in Table 2.3, DMon-guided optimizations outperform compile-time optimizations in 10/13 benchmarks. Crucially, static optimizations hurt performance in 5/13 cases due to being applied too broadly (with no runtime information), and therefore causing outcomes such as cache pollution and code bloat. DMon-guided optimizations always improve the performance. In 3/13 benchmarks where static optimizations outperform DMon-guided optimizations, the margin is $\leq$ 5% which can be reduced by reducing the incremental monitoring threshold (default, 10%) of

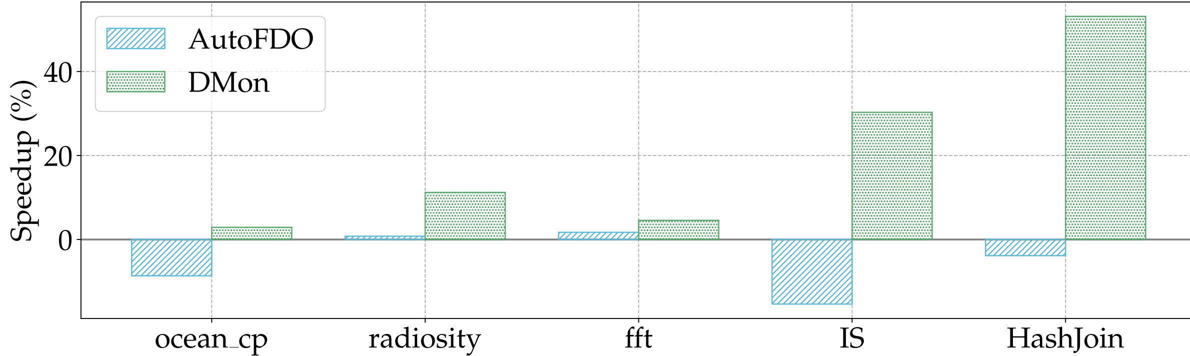


Figure 2.10: Speedup comparison to AutoFDO (All $\sigma < 0.09\mu$)

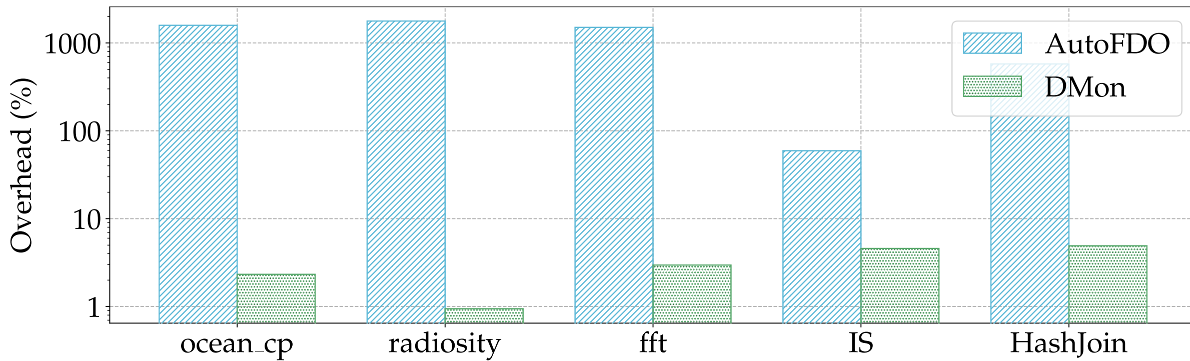


Figure 2.11: Overhead comparison to AutoFDO (All $\sigma < 0.07\mu$)

selective profiling.

Comparison against Google AutoFDO. We compare the speedup provided by DMon-guided optimizations to that of Google's AutoFDO [65], the state-of-the-art profile guided optimization technique. AutoFDO has limited data locality optimization capabilities [256]; our comparison is thus limited to five benchmarks for which AutoFDO can optimize locality.

We compare the speedup provided by DMon-guided optimizations to the speedup provided by AutoFDO in Fig. 2.10. As shown, DMon-guided optimizations provide better speedup than AutoFDO for all five benchmarks. This is because AutoFDO could only identify data locality problems that can be solved by performing direct prefetching optimizations. By contrast, DMon can identify other data locality issues that can be addressed by additional locality optimizations (*i.e.*, indirect prefetching, structure splitting, and structure merging).

For example, AutoFDO's direct prefetching slows down the execution of `IS` by 15%, while DMon-guided indirect prefetching provides a 30% speedup. Even for cases where both DMon and AutoFDO suggest direct prefetching (*e.g.*, `ocean_cp`), DMon-guided optimizations outperform AutoFDO, because, unlike AutoFDO, DMon provides hints as to where (*e.g.*, L1, L2, or L3) the cache line should be prefetched.

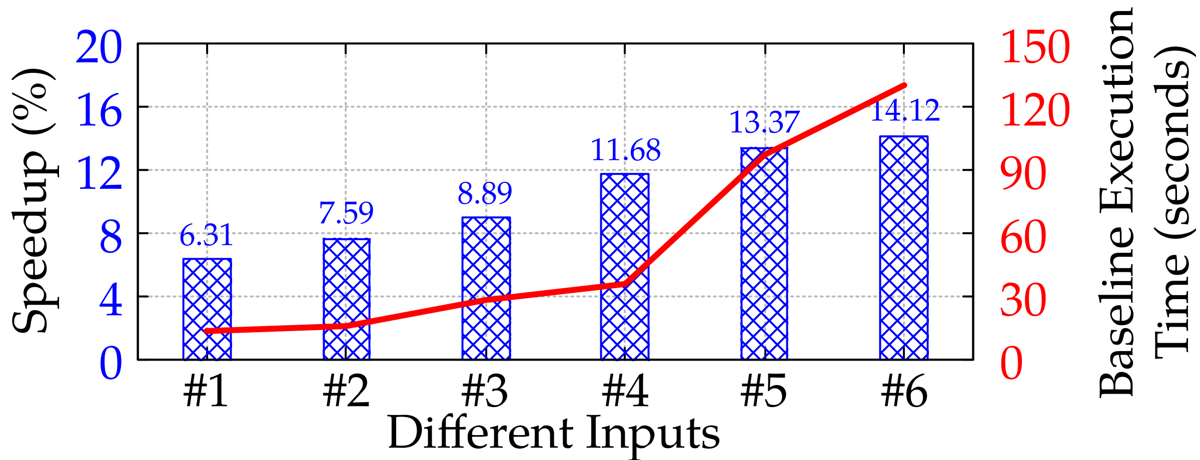


Figure 2.12: DMon-generated optimization after observing input #4 generalizes to unseen inputs (All $\sigma < 0.01\mu$).

We compare selective profiling overhead against AutoFDO's profiling overheads in Fig. 2.11. For the 5 benchmarks in this study, selective profiling incurs 3.3% mean overhead, whereas AutoFDO incurs 978% mean overhead, making the latter unsuitable for production use.

Generalization across program inputs. Profile-guided optimizations perform best when the application is optimized with a profile that is representative of the application's common behavior [288, 365, 65]. DMon-guided fixes also generalize if the program shows similar data locality behavior across different inputs. Therefore, we evaluate DMon's generality across different program inputs for 9 benchmarks. These program inputs vary widely both in terms of input size (from megabytes to gigabytes) as well as execution times needed to process the input (from seconds to minutes).

We report a detailed case study using the `radiosity` benchmark to determine how well the locality optimizations suggested by DMon generalize to different inputs. We choose this benchmark because the fix suggested by DMon is structure splitting—an optimization that modifies the data layout, and hence has the potential to be affected by changing program inputs. Fig. 2.12 shows the speedup provided by DMon-guided optimizations for `radiosity` for various input sizes.

Here, for brevity, we refer to different input sizes using "#1" through "#6". DMon only observes the execution for the randomly selected input #4. After observing input #4, DMon-guided optimizations are applied. Then, all inputs are rerun with the newly-optimized program, with the results of this run reported in Fig. 2.12. As shown, the optimization suggested by DMon generalizes well to other inputs, providing considerable speedups in each case. Longer executions that use *larger* inputs benefit more from optimizations.

Fig. 2.13 shows how DMon-guided optimizations improve data locality for unobserved inputs

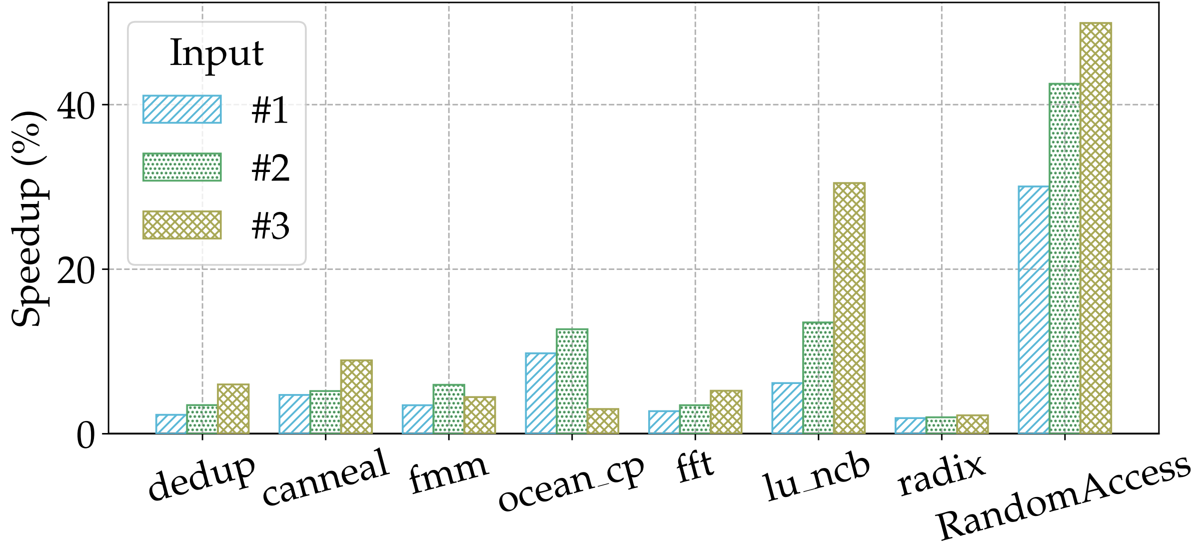


Figure 2.13: Input generalization (All $\sigma < 0.04\mu$)

of several other benchmarks. Here, we include all benchmarks with at least 3 inputs. Across all evaluation targets, we find that data locality behavior follows a similar trend for different inputs. Hence, DMon's fixes generalize to different inputs for these benchmarks.

Recompilation overhead. We evaluate the offline recompilation overhead while applying DMon-guided optimizations, though this does not impact the production overhead. We perform this experiment, because automated structure splitting and merging require pointer analysis, which is known to be expensive [214]. However, the specific pointer analysis we employ is flow- and context-insensitive and scales well [135].

Fig. 2.14 shows the offline compilation overhead incurred by our DMon-guided optimizations on top of the baseline compilation overhead (`clang`). On average, DMon-guided optimizations incur 72% more overhead. However, the optimization takes on average less than 7 seconds and is no longer than 26 seconds. Even for large applications (*e.g.*, PostgreSQL [348] code base has over 1M LOC), the analysis takes 307 seconds. For an offline process, we believe these durations are reasonable and on par with standard compiler transformations that use whole-program pointer analysis. Moreover, this is a one-time compile-time overhead and will be amortized for long-running applications (*e.g.*, data-center applications that are compiled once but run on thousands of servers for days). Finally, structure splitting and merging can be applied manually if the cost of pointer analysis is deemed prohibitive.

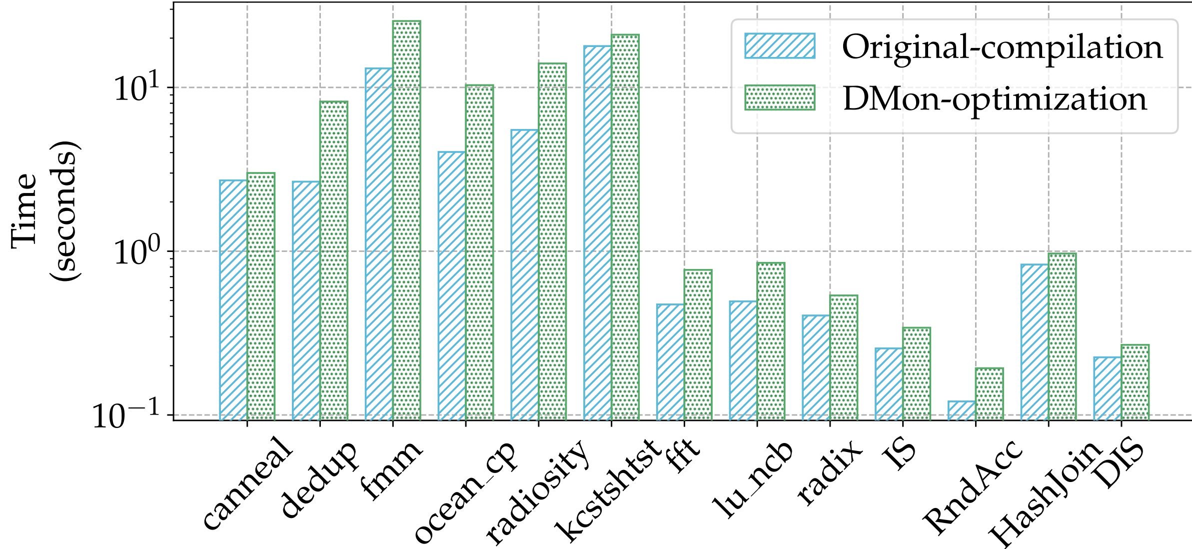


Figure 2.14: Overhead of DMon-guided optimizations compared to baseline compilation time ($\sigma < 0.1\mu$, log-scaled y).

2.6.3 Real-World Case Studies

We evaluate the applicability of selective profiling and DMon to large systems by studying (1) speedups provided by DMon-guided optimizations on PostgreSQL [292]—one of the most popular database systems, and (2) speedups achieved after manual repair of data locality problems detected by selective profiling for just-in-time (JIT) compiled real-world applications from the Renaissance benchmark suite [295].

PostgreSQL case study. We evaluate DMon's ability to improve the locality (and thereby performance) of `PostgreSQL` v11.2 [292], one of the most popular open-source database management systems. For this study, we run the popular TPC-H [85] queries on a 1GB database stored in PostgreSQL. We intentionally select the database size to fit in memory to ensure a memory-bound workload (instead of disk-bound one), as the vast majority of real-world databases fit in memory [291, 253].

To evaluate DMon, we profile PostgreSQL with DMon while serving all 22 TPC-H queries. For these queries, selective profiling incurs 1.2% average and 2.7% maximum overhead. For `PostgreSQL`, DMon identifies a locality problem in a function (`ExecParallelHashNextTuple`) that accesses the members `area` and `parallel_state` of structure `hashtable` [131]. DMon identifies that this memory access is the primary reason for poor data locality in 6 out of 22 TPC-H queries. Moreover, this memory access causes L2 and L3 cache misses for all 22 TPC-H queries. The cause of the locality problem in this case is pointer chasing. Structure merging automatically repairs this problem and speeds up all 22 TPC-H queries,

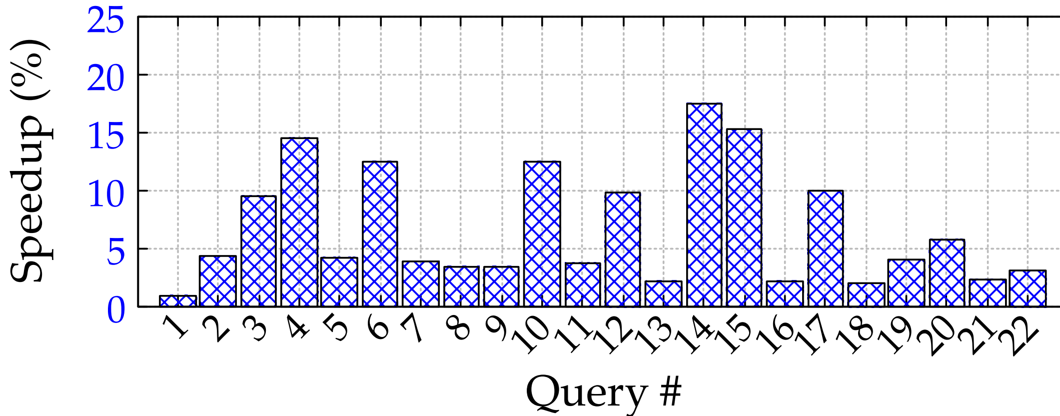


Figure 2.15: Speedup due to DMon-guided optimizations for 22 TPC-Hqueries on `PostgreSQL` (All $\sigma < 4.53\%$ of μ).

as shown in Fig. 2.15. The L3 cache misses in `PostgreSQL` are reduced by up to 22.11% (3.05% on average) and the latency of the 22 TPC-H queries are improved by up to 17.48% (6.64% on average). We also test optimized `PostgreSQL` based on DMon-profile on larger databases (10 and 100GB), where DMon improves the latency of the 22 TPC-H queries by 4.68% on average. For larger databases (10 and 100GB), the overall performance gain due to DMon's optimizations are comparatively less than (2% on average) that of smaller databases (1GB). That is because the performance of `PostgreSQL` for larger databases are primarily bottlenecked by storage I/O costs.

These results are particularly encouraging, considering that `PostgreSQL` is one of the most heavily-optimized codebases, having been improved by developers over the past 20 years. Most database developers hand-tune their code using the TPC benchmarks as regression tests (*i.e.*, their performance is best on TPC). This fact makes it even more promising that DMon-guided optimizations are able to improve the performance of these benchmark queries on a mature database system. We reported this data locality issue to the developers of `PostgreSQL` (for the version 11.2), which they have fixed since then.

Renaissance case study. A key advantage of just-in-time (JIT) compilation over ahead-of-time compilation (*e.g.*, Java vs. C++) is that JIT can apply dynamic optimizations—including limited data locality optimizations—using tiered compilation [245]. We compare selective profile-guided data locality optimizations to tiered compilation from OpenJDK [374] on real-world applications from the Renaissance suite [295]. For these applications, selective profiling incurs 2.2% average and 2.6% maximum overhead.

We use selective profiling to detect data locality issues in three Renaissance applications (jdk-concurrent `fj-kmeans`, apache-spark `page-rank`, and Scala `stm-bench7`). We omit other Renaissance benchmarks for which selective profiling does not find any data locality problems. Most

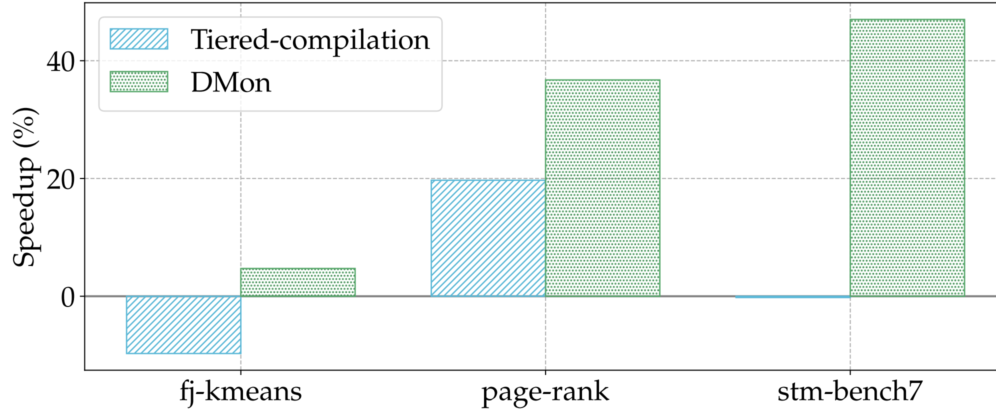


Figure 2.16: Speedup provided by selective profile-guided optimizations for just-in-time (JIT) compiled applications against tiered compilation (All $\sigma < 7.68\%$ of μ).

of the data locality issues found here corresponds to Java/Scala source code (we map binary instruction information back to Java code using perf-map-agent [170]) of Renaissance applications. Since currently DMon's optimizations only support C/C++ applications, we manually apply data locality optimizations to these applications. In all cases, we modify ¡10 LOC.

As shown in Fig. 2.16, selective profile-guided optimizations provide on average 26% and up to 47% more speedup than tiered compilation. This demonstrates that selective profiling is effective even for JIT-compiled applications.

Apart from these real-world case studies, we have also tested DMon on `Memcached` [117] and `RocksDB` [106] with YCSB benchmarks [80]. For these two applications, the individual pieces that make up the locality issues are relatively minor. Compiler-based data locality optimizations typically add extra instructions and logic in the code, which only helps when there are many cache misses causing slowdowns. For program statements responsible for a relatively small percentage of all cache misses (less than 5%), applying these optimizations do not provide any speedup, as the extra code and logic outweighs the benefits.

2.6.4 Sensitivity Analysis

We evaluate the impact of selective profiling's different parameters on effectiveness (coverage) and efficiency.

In-Production Monitoring Time-Slice. The granularity of the monitoring time-slice is a key design decision for selective profiling's incremental monitoring scheme (§2.3). Small time-slices allow selective profiling to identify locality problems for shorter-running applications, but also trigger frequent transitions during incremental monitoring and result in higher monitoring overhead. On the other hand, larger time-slices lower overhead but may fail to detect locality problems

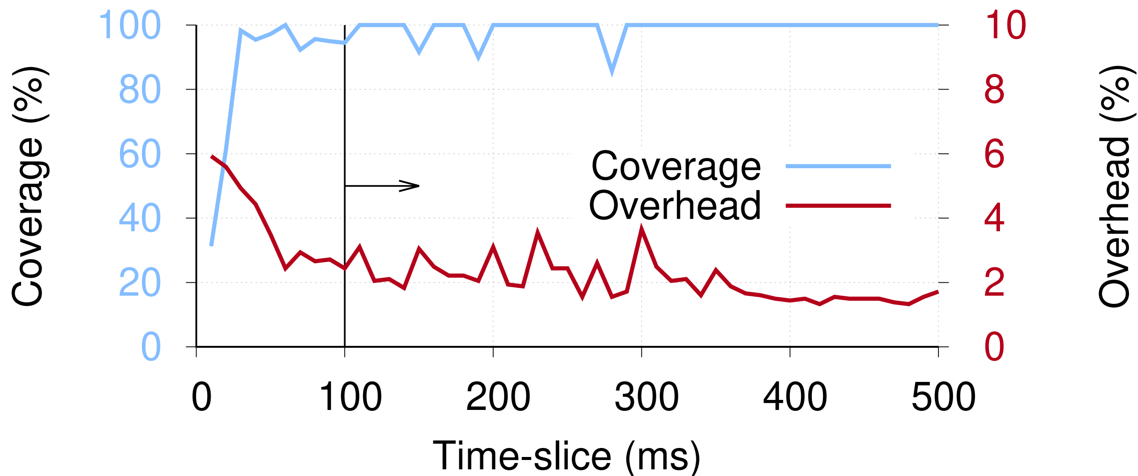


Figure 2.17: Effect of granularity of in-production time-slice on detection coverage and overhead (All $\sigma < 3.03\%$ of μ).

for shorter-running programs.

Fig. 2.17 shows the impact of the time-slice granularity on selective profiling's detection coverage (left y-axis) and overhead (right y-axis) for the benchmark, (`lu_ncb`). We vary the time-slice granularity from 10ms to 500ms (with 10ms increments) and measure selective profiling's coverage in detecting data locality issues and the associated performance overhead.

As shown in Fig. 2.17, selective profiling has lower coverage and higher overhead for smaller time-slices. As the time-slice granularity increases, selective profiling achieves greater coverage with lower overhead. Selective profiling's coverage is lower for smaller time-slices because selective profiling cannot monitor sufficient performance events in a small time slice. Beyond 100ms, both the coverage (99.07% on average with standard deviation of 3%) and the overhead (2.04% on average with standard deviation of 0.6%) lines flatten. Ergo, we set selective profiling's default time-slice as 100ms.

Incremental Monitoring Threshold. We vary the threshold of incremental monitoring (§2.3) from 1% to 50% and measure the coverage of data locality issues selective profiling detects for all 13 benchmarks in Table 2.2. 100% coverage is achieved when there is no incremental monitoring (*i.e.*, DMon continuously monitors events at the all levels of the locality tree). As shown in Fig. 2.18, selective profiling achieves greater than 80% coverage if the incremental monitoring scheme uses a threshold of ¡29%. Nevertheless, we set the default-threshold as 10%, as this threshold achieves 100% coverage.

In-Production Sampling Period. As described in §2.3, sampling period is a key design decision for selective profiling. Fig. 2.19 shows the impact of the sampling period on the coverage of locality

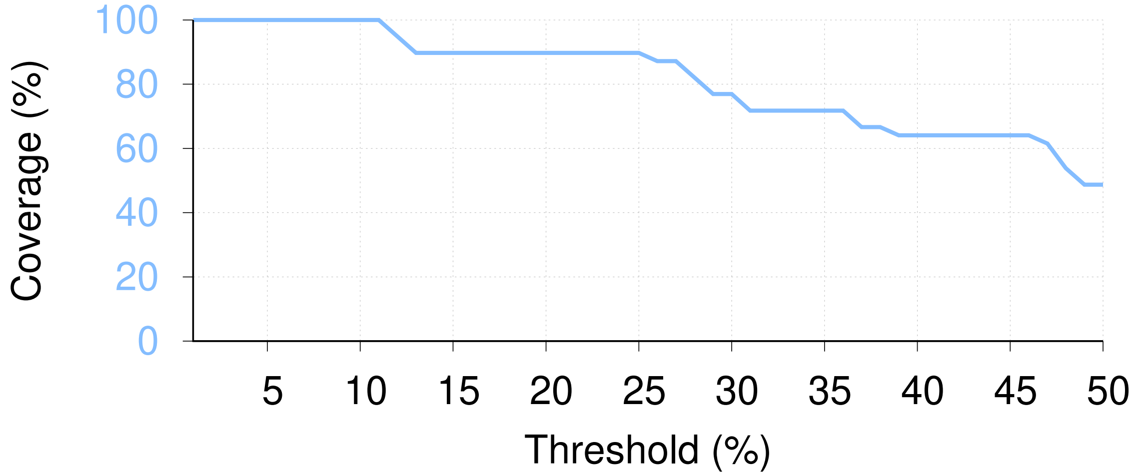


Figure 2.18: Effect of incremental monitoring threshold on the coverage of locality problems selective profiling detects across all benchmarks.

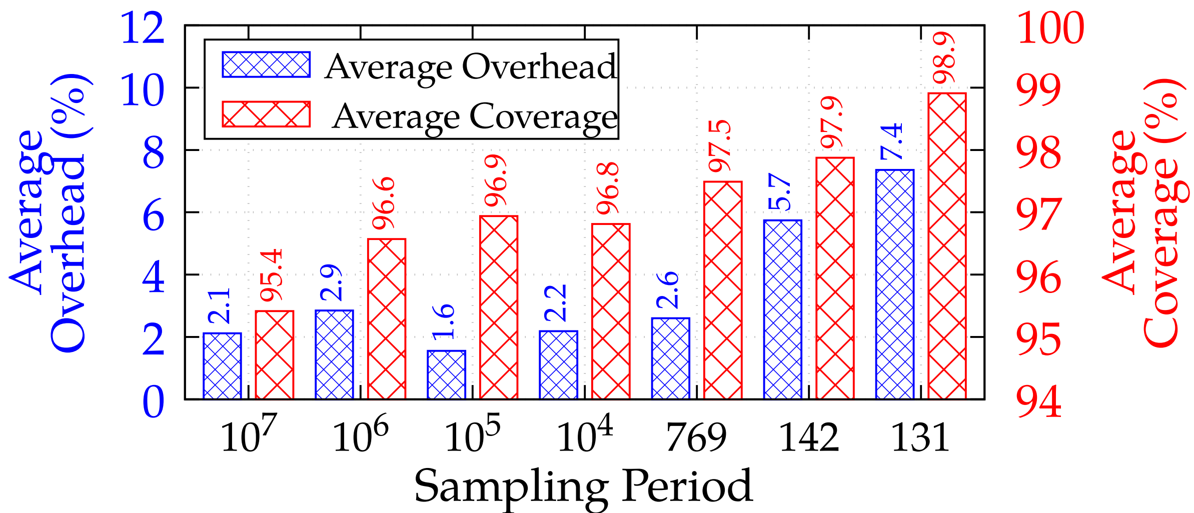


Figure 2.19: Effect of sampling period on the coverage of locality problems selective profiling detects and the average overhead across all benchmarks ($\sigma < 0.01\mu$).

issues selective profiling detects and its runtime overhead. We compute coverage with respect to the baseline coverage of 100%, achievable via the lowest possible sampling period offered by Linux `perf` (sampling every 100th event). A sampling period k on the x-axis means selective profiling will record one out of each k events. The left y-axis represents the runtime overhead and the right y-axis represents the coverage of locality issues selective profiling detects.

The overhead and coverage reported in Fig. 2.19 are arithmetic averages over all benchmarks. A smaller sampling period increases the overhead of selective profiling, but also increases coverage.

In our experiments, we chose a sampling period of 1000, which yields a high coverage of 97% with 2.6% overhead on average in layer 4 of selective profiling.

2.7 Related Work

DMon finds data locality problems with low overhead using selective profiling, identifies the root cause behind the problem, and guides optimizations to eliminate the problem. Existing profilers are not able to determine the root causes of data locality problems without incurring a high overhead.
Profilers. General-purpose profilers [370, 379, 220] report program hotspots without identifying the root cause behind performance problem. Consequently, recent studies propose specialized profilers to locate root cause for specific performance issues. Parallel profilers [119, 140, 169, 172] focus on critical path profiling to estimate potential performance gain [399, 88]. Synchronization profilers [29, 90, 402, 409] identify lock contention. Similarly, we design selective profiling as a specialized profiling technique for data locality. Selective profiling uses the APIs of a state-of-the-art profiler, Linux `perf`, and targets a subset of the events explored as part of the Top-Down [395]. Our main contributions over `perf` and Top-Down are: (1) full automation in profiling, (2) low-enough overhead for production deployment, (3) ability to automatically identify targeted optimizations based on the underlying performance problem.
Profile-guided data locality optimizations. Profile-guided approaches collect execution traces to identify where optimizations can be applied [70, 285, 232, 233, 183, 258, 199, 202]. State-of-the-art techniques [65, 274, 278, 126, 279] primarily address instruction locality. While prior work [311, 192, 204] also optimizes data locality, these solutions incur $>10\%$ profiling overhead. Selective profiling, however, incurs only 1.36% overhead on average (§2.6.1).
Static locality optimizations. Static approaches use complex analysis techniques to find opportunities to apply locality-improving transformations [63, 251, 335, 178, 97, 393, 55, 223, 66]. Alas, these techniques use compile-time heuristics to apply transformations, which can lead to sub-optimal speedups or even reductions in performance. To avoid these issues, we use application profiles collected by selective profiling to apply optimizations in a *targeted* manner, leading to better speedups and avoiding transformations which hurt performance.
Dynamic locality optimizations. There are several proposals for monitoring program execution and modifying program binaries to improve locality on the fly [366, 342, 102, 262]. These techniques require non-existent hardware support and incur high overhead (up to $6\times$ [366]). Just-in-time (JIT) compilation techniques [70, 145] provide limited data locality optimizations. On the other hand, DMon works with existing hardware, incurs negligible overhead, and guides optimizations that provide better speedup (16.83% on average).

2.8 Conclusion

Poor data locality is a major performance problem that hurt applications in production. Unfortunately, existing data locality profilers are not efficient enough to be deployed in production. This is limiting, since production profiles are difficult to replicate offline. We address this problem by selective profiling, a technique capable of discovering data locality problems with negligible overhead (on average 1.36%) in production. We also design DMon, which guides automatic and manual data locality optimizations based on profiles generated using selective profiling. For an extensive set of real-world applications and widely-used benchmarks, DMon provides up to 53.14% and on average 16.83% speedup for the cases where DMon applies targeted optimizations after detecting significant data locality problems.

CHAPTER 3

Huron: Hybrid False Sharing Detection and Repair

Writing efficient multithreaded code that can leverage the full parallelism of underlying hardware is difficult. A key impediment is insidious cache contention issues, such as false sharing. False sharing occurs when multiple threads from different cores access disjoint portions of the same cache line, causing it to go back and forth between the caches of different cores and leading to substantial slowdown.

Alas, existing techniques for detecting and repairing false sharing have limitations. On the one hand, in-house (*i.e.*, offline) techniques are limited to situations where falsely-shared data can be determined statically, and are otherwise inaccurate. On the other hand, in-production (*i.e.*, runtime) techniques incur considerable overhead, as they constantly monitor a program to detect false sharing. In-production repair techniques are also limited by the types of modifications they can perform on the fly, and are therefore less effective.

We[1] present Huron, a hybrid in-house/in-production false sharing detection and repair system. Huron detects and repairs as much false sharing as it can in-house, and relies on its lightweight in-production mechanism for remaining cases. The key idea behind Huron's in-house false sharing repair is to group together data that is accessed by the same set of threads, to shift falsely-shared data to different cache lines. Huron's in-house repair technique can generalize to previously-unobserved inputs. Our evaluation shows that Huron can detect more false sharing bugs than all state-of-the-art techniques, and with a lower overhead. Huron improves runtime performance by 3.82× on average (up to 11×), which is 2.11-2.27× better than the state of the art.

3.1 Introduction

Over the past decade, pervasiveness of parallel hardware has boosted opportunities for improving performance via concurrent software. Today, almost every computing platform—data centers,

[1] Some of the work in this chapter was performed in collaboration with Yifan Zhao, Gilles Pokam, Barzan Mozafari, and Baris Kasikci [203]. Therefore, I use the "we" pronoun in this chapter to acknowledge their involvement in this work.

personal computers, mobile phones—relies heavily on multithreading to best utilize the underlying hardware parallelism [20, 92, 74, 222, 36].

Alas, writing efficient, highly-multithreaded code is a challenge—subtle mistakes can drastically slow down performance. One prominent cause of this is cache contention, and true and false sharing are two of the most common reasons of cache contention [104]. True sharing occurs when multiple threads on different cores access overlapping bytes on the same cache line (*e.g.,* multiple threads accessing a shared lock object). False sharing, on the other hand, occurs when multiple threads access disjoint bytes on the same cache line [408]. In both cases, to maintain cache coherency [336], concurrent accesses will cause the cache line to go back and forth between the caches of different processor cores, thereby hurting performance.

In many cases, true sharing is intentional, *i.e.*, it may not be possible to prevent threads from sharing data on the same cache line. For example, developers intentionally share data among threads in order to implement a necessary functionality, such as a lock in a threading library or reference counters in a language runtime (*e.g.,* Java's Virtual Machine). Even when unintentional, developers can use existing tools (*e.g.,* profilers [146, 91, 231, 403]) to detect and fix true sharing.

False sharing, on the other hand, is more insidious. Developers may not be aware that threads accessing different variables at the source code level will end up on the same cache line at runtime. Therefore, false sharing is almost invariably unintentional: it is challenging for developers to determine the presence of false sharing while programming.

Due to the challenging nature of false sharing—and its devastating impact on performance (*e.g.,* over $10\times$ slowdown [52, 213])—there has been a lot of recent interest in automatically detecting and repairing it. Existing techniques for false sharing detection rely on static analysis [159, 75, 143], runtime monitoring [228, 408], compiler instrumentation [230], or hardware performance counters [104, 239]. In contrast, false sharing repair techniques rely on operating system support [228, 93], managed language runtime support [104], or custom in-memory data structures [239].

Alas, existing false sharing detection and repair techniques have limitations. In-house approaches [159, 75] use static analysis and compiler transformations to detect and repair false sharing, respectively. These techniques are therefore limited to eliminating false sharing for programs where the size and location of data elements can be determined statically. As a result, more recent false sharing detection and repair techniques work only in production. In-production approaches overcome the challenges of in-house techniques, but at the expense of suboptimal speedups [239], large runtime and memory overheads [228, 93, 230], narrow applicability [104, 93], and even memory inconsistency [228] (see §3.6 for details).

In this paper, we show that the common perception that in-house mechanisms are limited in terms of the types of programs for which they can detect and repair false sharing [228] is not

necessarily correct. We show that it is possible to determine the effect of different program inputs on a false sharing bug, even after observing the bug for a single input.

Relying on the above observation, we present Huron, a hybrid in-house/in-production false sharing detection and repair system. For all false sharing bugs that can be detected in-house, Huron's novel algorithm generates a repair that can, in many cases, generalize to different program inputs. For false sharing instances that cannot be detected in-house, we leverage an existing in-production false sharing detection and repair mechanism [93], which we improved to only detect previously-unobserved false sharing instances with greater efficiency (*i.e.*, by caching the already-detected ones). Our insight is that, in many cases, developers will have in-house test cases that exercise the most performance-critical paths of their programs, which will allow our in-house false sharing detection and repair to be effective.

Performing false sharing detection and repair in house allows us to devise a novel repair mechanism that works by transforming the memory layout, which would have been otherwise too expensive to use in production. *The key insight behind our in-house false sharing repair is to group together data that is accessed by the same set of threads, and thereby shift falsely-shared data to different cache lines (i.e., eliminate false sharing).*

Despite repairing most false sharing in-house, we empirically show that, in many cases, Huron's repairs generalize to different inputs (*e.g.*, configuration parameters, thread counts, etc.), because the relation between a program's input and false sharing can usually be determined accurately using Huron's conservative static analysis. Huron can then use this relation to generate a fix that generalizes to any input.

In addition to eliminating false sharing, Huron's memory grouping improves spatial locality.

In summary, we make the following contributions:

- We present Huron, a hybrid in-house/in-production false sharing detection and repair technique. Huron's in-house technique uses a novel approach to eliminate false sharing by grouping together memory that is accessed by the same set of threads.
- We show that Huron can generate false sharing fixes that generalize to different program inputs.
- We show that Huron eliminates false sharing using benchmarks and real programs, delivering up to 11$\times$ (3.82$\times$ on average) speedup. Compared to the state of the art [228, 93], Huron delivers up to 8$\times$ (2.11-2.27$\times$ average) larger speedup, on average 33.33% higher accuracy, and up to 197-377$\times$ (on average 27-59$\times$) lower memory overhead.

3.2 Background and Challenges

In this section, we first discuss the key challenges faced by false sharing detection and repair techniques. We then briefly describe how Huron addresses each of these challenges.

3.2.1 Effectiveness

The effectiveness of a false sharing repair[2] mechanism is the extent to which it can improve a program's performance.

In-production false sharing repair mechanisms modify the executable on the fly [228, 104, 93, 239], meaning they are limited in the extent of modifications they can perform and are less effective than in-house repair, as we show in §3.5.4. These techniques reduce false sharing by either splitting the falsely-shared data between different pages [228, 93] or by using special runtime support [239].

False sharing can be repaired more effectively if one can *surgically* modify a program's code and data at a fine granularity. A common and effective repair technique is to simply pad a cache line with dummy data in order to move the falsely-shared data to separate cache lines [354]. In fact, some in-production repair approches also use this technique [104]. However, these techniques are only applicable to managed languages (*e.g.,* Java), where programs pause at well-defined points (*e.g.,* garbage collection), thereby allowing for the code and memory layout to be restructured efficiently.

Even when existing approaches are able to repair false sharing, we show in §3.5.4 that, in many cases, they are much less effective than Huron (up to $11\times$). In particular, by performing most of the false sharing repair in house, Huron achieves more speedup. Finally, since Huron does not rely on dummy padding, it can even outperform manual false sharing repair in many cases (7 out of 9, as we show in §3.5.4).

3.2.2 Efficiency

A false sharing detection and repair mechanism is considered *efficient* if its runtime performance overhead is low.

In-production techniques monitor the program for false sharing instances to fix them on the fly, thus, incurring considerable runtime overhead (*e.g.,* up to $11\times$ [228]).

The efficiency of in-production false sharing detection techniques is further hindered by the fact that different program inputs may require detecting different instances of the same false sharing bug and generating a new repair strategy, both of which are costly. Execution #1 in Fig. 3.1 has a false sharing bug where threads t_1 and t_2 each access 60 bytes of data, and therefore share the last 4 bytes of cache line #1. Execution #2 shows that this false sharing instance can be repaired by padding each cache line with 4 bytes of data, which will force t_1 and t_2 to access their data on separate cache lines. On the other hand, in Execution #3, which is the result of a different input, each of the two threads accesses 30 bytes of disjoint data residing on a single cache line of 64 bytes. However, the 4-byte padding is not enough to shift the last 30 bytes to a separate cache line.

[2]Since detection is a binary prediction, the effectiveness of a *detection* mechanism is the same thing as its accuracy.

Therefore, fixing the false sharing in this instance will require a 34-byte padding (or an additional 30 bytes), as shown in Execution #4.

To alleviate the overhead of in-production false sharing detection and repair, Huron performs as much of its detection and repair in house as possible, *e.g.*, by using test cases, etc. Consequently, Huron's in-production detection and repair is triggered less frequently than previous techniques and incurs less overhead (see §3.5.9). Furthermore, in many cases, Huron can generate an input-independent repair strategy that generalizes to multiple inputs (3.3.4).

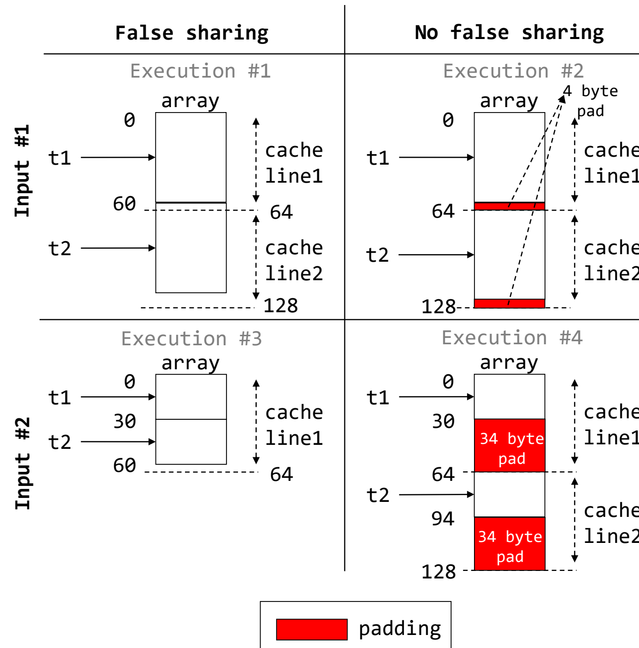


Figure 3.1: Input-dependent false sharing repair

3.2.3 Accuracy

The accuracy of a detection technique is the extent to which it can detect correct instances of false sharing (true positives) without flagging incorrect instances (false positives).

In theory, it is possible to build an in-production false sharing detector that does not suffer from false positives. However, since this is costly, all state-of-the-art false sharing detection strategies resort to sampling hardware events [93, 104] that are indicative of false sharing (*e.g.,* Intel HITM [83]) or use approximate algorithms [228].

By combining in-house and in-production false sharing detection, Huron achieves the best of both worlds. As shown in §3.5, Huron is not only more accurate than state-of-the-art detection approaches, but it is also more efficient.

3.3 Design

In this section, we describe the design of Huron, our hybrid in-house/in-production false sharing detection and repair system. Huron first detects and repairs false sharing in house using developer test cases. For false sharing instances that may not be detected or fixed in house (*e.g.,* due to thread nondeterminism or change of input), Huron uses its optimized version of an existing in-production technique [93].

Fig. 3.2 shows various components in Huron's design. Steps ①–④ occur in-house. In step ① (§3.3.1), the source code of the target program is fed into an instrumentation pass, responsible for instrumenting the program with false sharing detection code. In step ② (§3.3.2), Huron's in-house detection component detects false sharing using the instrumentation from the previous step. In step ③, Huron saves metadata (*e.g.*, program counter etc.) regarding the detected false sharing instances in a cache that is used to speed up in-production detection and repair. In step ④ (§3.3.3), the detected false sharing instances are used to perform memory layout transformations. The repair mechanism in this step groups together data that is accessed by the same set of threads, while separating falsely-shared data into different cache lines. Another key sub-step of the memory layout transformation is a special compiler pass (§3.3.4) that produces a generic false sharing repair strategy that works for multiple program inputs.

Steps ⑤ – ⑧ (§3.3.5) occur in production. In step ⑤, the program is deployed, while Huron performs its in-production false sharing detection. Moreover, Huron uses a cache of already-detected (in-house) false sharing instances – in step ⑥ – to reduce overhead. When Huron detects a new false sharing instance in production, it fixes it by separating falsely-shared data into different pages using an existing tool [93] in step ⑦. Finally, in step ⑧, Huron saves the false sharing instances it detected in production, so that they can be repaired more effectively using the memory layout transformations the next time the program is built and deployed.

3.3.1 Instrumentation Pass

Huron uses a compiler pass to instrument memory access and allocation instructions. Similar to all prior work, Huron's detection is geared towards detecting false sharing of global data and dynamically-allocated data. Huron does not monitor stack data for false sharing. While it is possible–although considered poor practice–for threads to share data through the stack, we have not observed this in practice.

Huron instruments all heap and global memory accesses, which is necessary for accurate detection. These include load and store instructions (including atomic load/store) as well as atomic exchange instructions. At runtime, the instrumentation logs the target size and the memory address

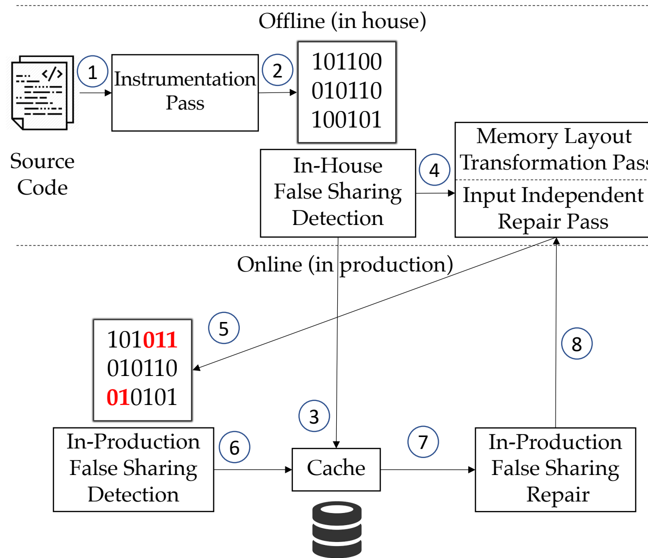


Figure 3.2: High-level design of Huron

of the load/store as well as the program counter of the instruction.

Huron also instruments all memory allocation instructions in order to collect the information necessary for generating false sharing fixes that generalize beyond the inputs observed in house. In particular, Huron's instrumentation inserts external function calls that log the program counter of the memory allocation instructions, as well as the start and end addresses of the allocated memory.

3.3.2 In-House False Sharing Detection

To ensure accurate detection, for each thread, Huron tracks how many bytes of data are accessed on which cache lines. If multiple threads access disjoint data on the same cache line at any point, Huron flags this as a false sharing instance.

The listing in Fig. 3.3 shows a simplified example–adapted from `histogram` [300]–that suffers from false sharing when multiple threads spawned from `main` (thread creation code omitted for brevity) concurrently execute `calc_hist`, which increments thread-specific counters for pixel values, red and blue (green omitted for brevity). Since pixel values vary from 0 to 255, there are 256 counters (of type int) for each color. Each thread executing the function iterates over a portion of the image pixels (specified by pointers `begin` and `end`) to retrieve and increment each counter. Fig. 3.4 shows the memory layout of an object of type `global_t` for a two-thread execution, *i.e.*, `N_THREAD = 2`. The `inputs` array is aligned at the beginning of a cache line and spans 24 bytes. Following `inputs` are the four subarrays `red[0]`, `red[1]`, `blue[0]`, `blue[1]`. The subarrays `red[0]` and `blue[0]` are accessed by thread 1 and the subarrays `red[1]` and `blue[1]` are accessed by thread 2. The dashed boxes denote the cache lines where subarrays both reside and cause

```
1 struct global_t {
2   char *inputs[N_THREAD+1];
3   int red[N_THREAD][256];
4   int blue[N_THREAD][256];
5 };
6 global_t *global;
7 main(...) {
8   global = malloc(sizeof(global_t));
9   for(int i=0;i<N_THREAD;i++) {
10    for(int j=0;j<256;j++) {
11      global->red[i][j]=0;//Location 1
12      global->blue[i][j]=0;//Location 2
13    }
14  }
15 }
16 void calc_hist(int tid) {
17   char *begin=global->inputs[tid];
18   char *end=global->inputs[tid+1];
19   for (char *p=begin; p<end; p+=2) {
20     global->red[tid][*p]++;//Location 3
21     global->blue[tid][*(p+1)]++;//Location 4
22   }
23 }
```

Figure 3.3: Listing with false sharing that occurs when multiple threads execute the lines 20 and 21 in `calc_hist`.

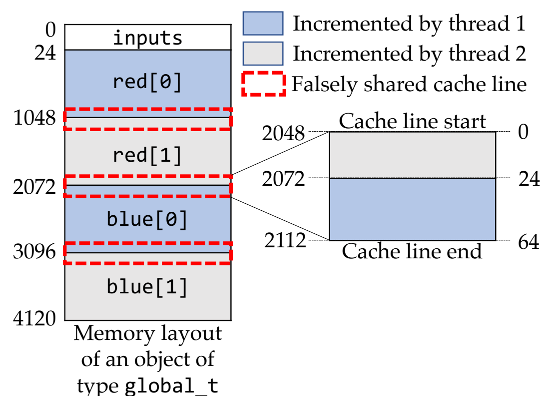


Figure 3.4: Memory layout of `global_t` data structure described in Fig. 3.3 and how the program suffers from false sharing due to thread 1 and thread 2 both incrementing non-overlapping values on the same cache line.

false sharing when thread 1 and thread 2 access the lines from different cores.

To detect false sharing in house, Huron relies on testing workloads (*e.g.,* unit tests, integration test, stress tests). The rationale behind this design decision is that most of the time, developers already have test cases that cover the most performance-critical paths of their programs, which would allow Huron to detect and repair most false sharing bugs in-house. Those that are missed in house are detected and repaired by Huron in production (see §3.3.5).

During in-house false sharing detection, Huron records and computes certain information to account for dynamic program behaviors that vary across different runs in production. For instance, falsely-shared data will likely reside in different memory locations each time the program is run (*e.g.,* due to dynamic memory allocation [76] and ASLR [323]). Therefore, Huron needs to uniquely identify the location of false sharing during the detection phase to be able to repair it during any in-production run. Furthermore, Huron needs to uniquely identify the instructions involved in false sharing in order to modify their access offsets during repair. Finally, since Huron repairs false sharing by grouping together memory locations that are accessed by the same set of threads, it needs to track which threads access which part of the memory. To achieve these goals, Huron tracks and computes the following information:

1. A unique identifier of each *program location* accessing memory. This is a 3-tuple `(file name, line number, execution count)`, which uniquely identifies the location even if it is inside multiple loops. For brevity, in Fig. 3.5, step 1, we show the unique program locations as 1, 2, 3, 4 which correspond to lines 11, 12, 20, and 21 in Fig. 3.3.
2. A unique identifier of each *memory region*, defined as a combination of a memory allocation site and an access offset range. The allocation site is a 3-tuple `(file name, line number, execution count)` that uniquely identifies the allocation operation. In the example of Fig. 3.3, this would be `(histogram.c, line 8, 0)`. Huron converts the memory addresses accessed by each instruction into a range of offsets with respect to the allocation site. These offsets are calculated by subtracting the base address returned by the memory allocator from the accessed memory address. In Fig. 3.5, step ①, we omit the allocation site tuple and only show the memory offset ranges for brevity.
3. A *thread ID*, as shown in the last column of step ①, Fig. 3.5, where thread 0 is the main thread, and threads 1 and 2 are the worker threads executing `calc_hist` from Fig. 3.3.

These three pieces of information are used to produce the thread access bitmap (*i.e.,* the set of thread accesses) of each memory region (step ②, Fig. 3.3). Using this bitmap, Huron identifies and repairs false sharing (see §3.3.3).

Effect of Detection Window Granularity. Huron considers the entire execution of a program as a single time window when detecting false sharing instances. It is therefore possible for Huron to miss certain instances of false sharing (*i.e.*, incur false negatives). To see why, consider two

disjoint time windows w_1 and w_2, where the entire execution runs for $w_1 + w_2$. Also consider two threads t_1 and t_2 accessing a 64 byte cache line l. Let's assume that in time window w_1, t_1 accesses the first 32 bytes of l, and t_2 accesses the last 32 bytes. Therefore in w_1, t_1 and t_2 are involved in false sharing. Now let's assume that in time window w_2, t_1 accesses the last 32 bytes of l, and t_2 accesses the first 32 bytes. Similarly, in w_2, t_1 and t_2 are involved in false sharing. However, if we consider the entire execution window $w_1 + w_2$, since both t_1 and t_2 access the first and last 32 bytes of l, they are truly sharing the cache line, hence there is no false sharing. Although it might seem useful to consider a fine-grained time window based on this example, this approach also suffers from false negatives, because it may not be possible to observe accesses involved in false sharing using a short window. Huron allows a developer to specify a detection time window. In §3.5.11, we show that using the entire exection as a single time window for false sharing detection is more effective.

3.3.3 Memory Layout Transformation Pass

We now describe how Huron repairs false sharing, assuming that program inputs and thread counts do not alter the false sharing behavior. We describe how Huron accounts for different inputs and thread counts in the next section.

Algorithm 1 describes a simplified version of Huron's memory layout transformation technique. The function TRANSFORM-LAYOUT takes a list of memory bytes M as input. Each byte $m \in M$ has an attribute $m.bitmap$ denoting the IDs of threads which accessed this memory byte during in-house false sharing detection. The algorithm populates Q, which maps a thread ID bitmap (*i.e.*, a set of thread accesses) to a list of all the memory bytes accessed by the threads described via the bitmap (Line 1-5). For each memory byte m accessed by the same set of threads b, the algorithm sequentially assigns an offset (1 byte) in memory. The hashmap T keeps track of each new offset. After all memory bytes m with the thread access bitmap b are assigned an offset, the algorithm computes the new offset i to be the next multiple of cache line size CLSIZE (Line 13). This ensures that the next byte with a different thread access bitmap will be placed in a different cache line. Since false sharing occurs among memory accesses with different corresponding bitmaps, the algorithm eliminates any potential false sharing.

For example, step ② in Fig. 3.5 shows how the threads from the example in Fig. 3.3 access various memory regions. The thread access bitmap in step ② shows where false sharing occurs with dashed boxes. Since memory regions [24-1048) and [1048-2072) share a cache line between offsets [1024-1098), and are accessed by different threads (i.e, with different thread bitmaps, where $011 \neq 101$), Huron detects false sharing. Huron then groups together memory regions with the same thread access bitmap as shown in step ③. Finally, Huron restructures the memory layout by

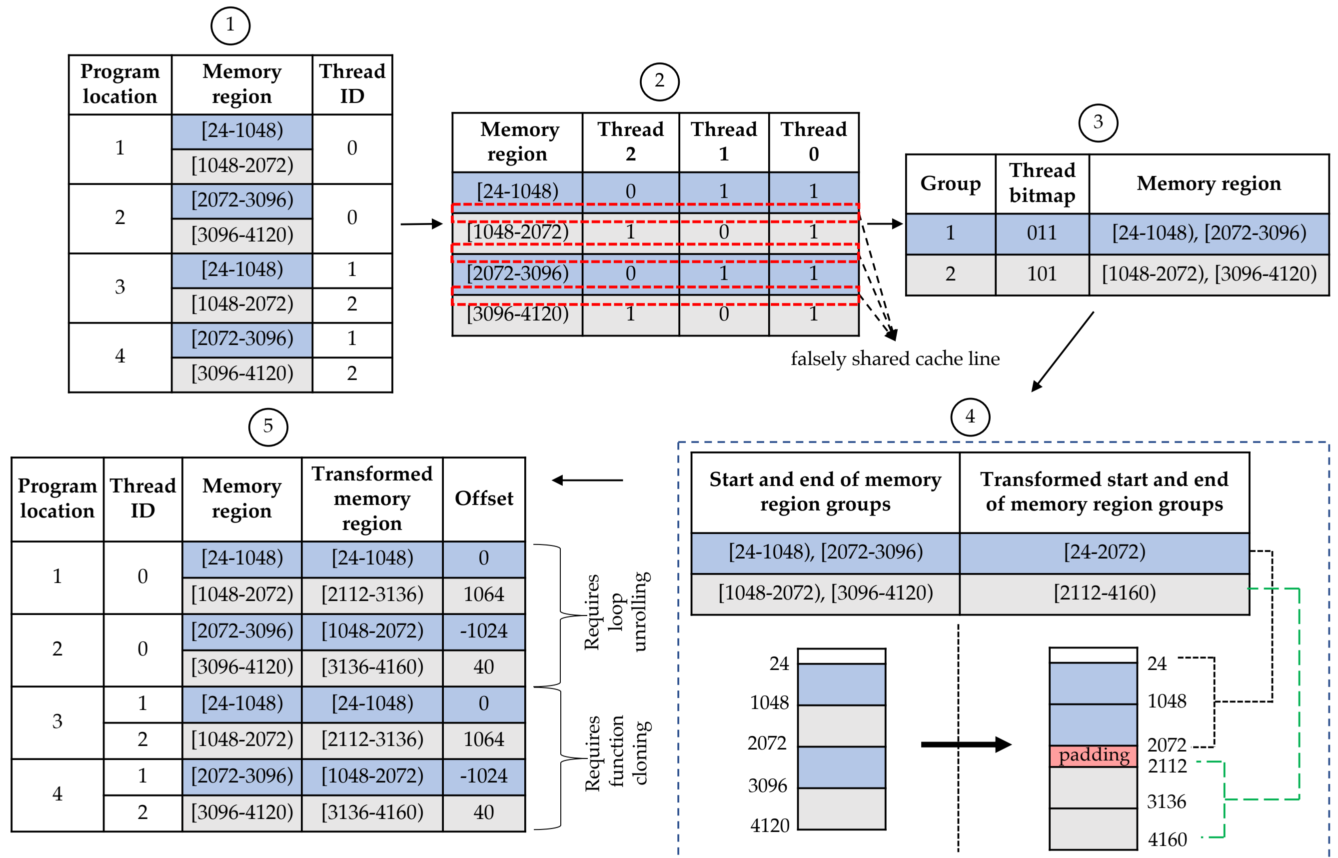


Figure 3.5: In-house false sharing repair via memory layout transformations for the example in Fig. 3.3. Memory region $[x-y)$ denotes bytes starting from x (inclusive) up to byte y (exclusive). For simplicity, we have omitted access to memory region $[0-24)$, *i.e.*, variable `inputs`. Cache line size is 64 bytes.

Algorithm 1 False sharing repair via memory layout transformation

TRANSFORM-LAYOUT(M)

```
1:  Q ← Hashtable()
2:  for each m ∈ M do
3:      if Q[m.bitmap] = NIL then
4:          Q[m.bitmap] ← List()
5:      Q[m.bitmap].insert(m)
6:  i ← 0
7:  T ← Hashtable()
8:  for each b ∈ Q.keys do
9:      for each m ∈ Q[b] do
10:         T[m] ← i
11:         i ← i + 1
12:     if i%CLSIZE ≠ 0 then
13:         i ← i + (CLSIZE −(i%CLSIZE))
14: return T
```

placing memory regions in consecutive cache lines, as shown in step ④.

It is still possible for false sharing to occur between the end of a group of memory regions and the start of the next group. For instance, in Fig. 3.5, step ④, since the transformed groups share a cache line, Huron inserts 40 bytes of padding between the ranges [24-2072) and [2112-4160). As opposed to manual techniques that introduce dummy padding, Huron uses existing data in the program (*e.g.,* from the heap or the stack), thereby outperforming manual false sharing repair in most cases, as we show in §3.5.3.

Then, memory layout transformation updates instructions to access memory in the new layout. Huron uses two techniques for this in step ⑤: (1) *loop unrolling:* For some program locations, Huron needs to insert different memory access offsets for instructions for different iterations of a loop. For instance, the offsets that Huron needs to insert for program location 1 in Fig. 3.3 are 0 (iteration, `i=0`) and 1064 (iteration, `i=1`). (2) *function cloning:* For some program locations, Huron needs to add different offsets for different threads (example, location 3 and 4 in Fig. 3.5). Hence, Huron clones the function containing the program location and adds different offsets for each clone. For instance, `calc_hist` function contains the program location 3 where the offsets are 0 (Thread 1) and 1064 (Thread 2). As we explain in §3.3.4, Huron can adjust the offsets to account for thread counts in different executions.

Huron has to ensure correct and unmodified memory access semantics for the program after memory layout transformations. For this, Huron uses a complete, interprocedural, inclusion-based pointer analysis [30] to determine all the instructions that can access the modified layout. Huron instruments these instructions to (1) check whether they are accessing the new layout at runtime,

and if so, (2) adjust the memory access offsets of the instructions accordingly. Alas, this analysis can have false positives, *i.e.,* instructions that Huron incorrectly considers as accessing the new layout. A large number of false positives will result in a large number of runtime checks, which might nullify Huron's performance improvements. To alleviate this, Huron only fixes false sharing if an instruction that needs to be modified took less than 1% of the total execution time during detection.

3.3.4 Input-Independent False Sharing Repair

In many cases, Huron can generalize its false sharing repair strategy to different program inputs after having detected false sharing for a single input. For such repairs, Huron does not need to rely on in-production false sharing detection and repair. Huron performs a *static range analysis* pass during layout transformation to compute the maximal range of memory that a given instruction can access, regardless of the program input. Consequently, Huron generates a memory layout transformation that will work for different inputs.

Algorithm 2 describes Huron's static range analysis, which leverages the type system to determine whether an operand of a memory access instruction (*e.g.,* store, load, atomic store, etc.) is an array type, and hence has fixed size. If that is the case, Huron determines the maximum memory range that the instruction can access based on its size.

FINDMAXRANGE takes in as input an operand `Ptr` of a memory access instruction. The algorithm computes and returns the maximum range that the instruction can access by iterating over all aliases, `a`, of `Ptr` (Line 4). If the alias, `a`, is a function argument (Line 5), then all calls to this function, `f` invoked with argument `x`, are analyzed to determine the maximum range of `Ptr` (Line 9). All the ranges computed in each function are appended to the list `ranges` (Line 10). The algorithm then returns the widest possible range (Line 11). If `a` is found to be allocated via a `malloc` call (Line 12), then the algorithm returns `[S, S+R]` as the maximum range. Finally, if `a` is found to be derived using pointer arithmetic (Line 14), then `S` is recalculated based on `a`'s base pointer, `base`. If `base` is of array type, then `R` is multiplied by the number of elements in the array. If `Ptr` is found to be none of function argument, `malloc` call, or pointer arithmetic, then the algorithm will return an empty range (*i.e.*, the fix can not be input-independent).

To understand how Huron's input range identification pass works, consider the statement `global->red[tid][*p]++;` on line 20 in Fig. 3.3, which increments a counter of how many pixels of an input image have matching *red* values. Since it is possible for an input to not contain all the 256 *red* levels, it is possible that during in-house false sharing detection, only `red[0][30]-red[0][56]` and `red[0][95]-red[0][197]` are accessed (assuming `tid = 0`).

Algorithm 2 Maximal memory range detection for input-independent repair

FINDMAXRANGE(Ptr)

```
1:  R ← size of element type of Ptr                          ▷ size of the range
2:  S ← 0                                                    ▷ starting offset of the range
3:  while True do
4:      a ← next alias of Ptr
5:      if a is a function argument then
6:          ranges ← []
7:          f ← function of a
8:          for each call to function f invoked with argument x do
9:              for each range (l, r) in FINDMAXRANGE(x)) do
10:                 append (l + S, r + S + R) to ranges
11:         return [FINDMIN(ranges), FINDMAX(ranges)]
12:     if a is a call to malloc then
13:         return [(S, S + R)]
14:     if a is derived from pointer arithmetic then
15:         base ← base pointer of a
16:         baseT ← type of base
17:         if baseT is an array type then
18:             n ← number of elements in baseT
19:             R ← R × n
20:         Ptr ← base
21:     else
22:         return []
```

Without input-independent false sharing repair, Huron would only perform memory layout transformations in this range and fail to repair false sharing for the rest of the array. However, with input-independent analysis, Huron discovers the user code can access `red[0][0]-red[0][255]`, and generates a fix for the entire range.

If Huron statically determines a linear relationship between thread counts and the offsets in the transformed layout, it will parametrize the offset to be a function of the thread count. This allows Huron's fixes to generalize to different thread counts. For instance, the sub-array `blue[0][256]` (of Fig 3.3) has a parametrized offset function `(-(N_THREAD-1)*1024)`, allowing the offset to be changed from (-1024) to (-2048) when `N_THREAD` changes from 2 to 3.

Huron can generate input-independent false sharing repair for many programs, as we show in §3.5.5. However, this is not always possible. For example, the type information may be lost due to excessive pointer casts, or certain ranges may not be determined statically. In such cases, it relies on in-production false sharing detection and repair.

3.3.5 In-Production False Sharing Detection and Repair

In production, Huron deploys the program that was repaired in house and leverages a modified version of an existing in-production false sharing detection and repair tool, namely TMI [93]. In a nutshell, TMI works by monitoring hardware performance counters (i.e, HITM), which was shown in prior work [239, 93] to be indicative of false sharing.

A surge in HITM counts triggers TMI's false sharing detection. Huron's metadata cache of previously-detected false sharing instances (containing program locations, memory offsets, type of false sharing etc.) speeds detection up. We show in §3.5.9 that Huron's modified in-production false sharing detection technique is on average 2.1× faster than TMI.

If Huron discovers a false sharing bug in production, it uses TMI to create a *temporary* fix by converting threads to processes. However, as we demonstrate in our evaluation (§3.5), such a repair mechanism may not be effective or efficient. More specifically, TMI mistakenly treats true sharing as false sharing for a number of benchmarks and moves truly-shared data to different pages to ”repair” this mistakenly-detected (i.e., false positive) false sharing instance. Even though TMI employs a memory-page-merging technique that ensures such false positives do not impact correctness (§2.2 of [93]), TMI's ”repair” degrades performance due to the expensive nature of the merging technique. To overcome these challenges, Huron keeps a record of the detected false sharing instance for in-house repair, which it will attempt next time the program is built and deployed in production.

3.4 Implementation

We implemented Huron in 5,682 lines of C++ code. Huron uses LLVM [215] for instrumentation, analysis, and memory layout transformations. The static range analysis pass (§3.3.4) leverages the language type system. To infer the type of an object pointed to by a pointer, the pass traces the pointer back to the instruction where the memory for the object was allocated. Huron does this by recursively iterating over the use-def chain that leads up to the allocation site. We also integrated an existing Andersen-style alias analysis [67] into Huron for its input-independent repair pass.

Huron's in-house runtime tracks thread creations, memory allocations, and loads/stores using the instrumentation code as well as a shim library that intercepts and logs memory allocation and thread creation operations (*i.e.*, via `LD_PRELOAD`). Huron is open-sourced [103].

3.5 Evaluation

In this section, we answer several key questions:

- **Accuracy:** How accurately can Huron detect false sharing compared to the state of the art (§3.5.2)?
- **Effectiveness:** Can Huron repair more false sharing bugs than state-of-the-art tools? (§3.5.3)? How does the speedup provided by Huron compare to manual and state-of-the-art false sharing repair tools' speedup (§3.5.4)? How effective is Huron's input-independent false sharing repair (§3.5.5)? How the quality of in-house test cases affects Huron's effectiveness (§3.5.6)?
- **Efficiency:** What is the overhead of Huron's repair mechanism compared to the state of the art (§3.5.7)? How much overhead does Huron's in-house detection incur (§3.5.8)? How beneficial is Huron's false sharing cache in speeding up in-production detection (§3.5.9)? To what extent Huron's in-house component improves the efficiency of Huron's in-production component (§3.5.10)? How the false sharing detection time window granularity affects Huron's repair speedup (§3.5.11)?

3.5.1 Experimental Setup

Software. All experiments are conducted in Ubuntu 16.04, with Linux kernel version 4.4.0-127-generic using LLVM's front-end compiler clang 7 [215].

Hardware. We use a 32-core Intel E5-2683 machine with 128 GB of RAM.

Baselines. We compare Huron to the following state-of-the-art techniques:

1. **Sheriff** [228] is an in-production framework consisting of two tools: Sheriff-Detect and Sheriff-Protect. Sheriff-Detect tracks updates to a cache line by multiple threads to detect false shar-

ing. Sheriff-Protect repairs false sharing by transforming threads into processes since, unlike threads, processes do not share the same address space.

2. **TMI** [93] is another in-production detection and repair mechanism. TMI monitors the surge in hardware events (*i.e.*, Intel HITM) to trigger its false sharing detection algorithm. TMI also uses thread-to-process transformation to eliminate false sharing.
3. **Manual** is a baseline where a human programmer repairs false sharing using dummy data padding to separate falsely-shared data onto different cache lines. Although laborious, manual repair can provide significant speedups. In fact, state-of-the-art tools—*i.e.*, Sheriff and TMI—consider the speedups provided by manual repair an upper bound (which we show in §3.5.4 not to be the case).

We do not include in-house false sharing detection and repair baselines in our evaluation, since these techniques [159, 75, 358] are targeted at specific applications and do not work well for the range of benchmarks we look at. For example, Jeremiassen et. al. [159] use an analysis that does not support complex access patterns, like accessing an array using values from another array as indices, just like `histogram` does.

Benchmarks. We use well-known benchmarks from the popular Phoenix [300] and PARSEC [50] suites, which have been used by many previous techniques for false sharing detection and repair [230, 228, 408, 269, 239, 229, 93]. We omit Parsec and Phoenix benchmarks that do not suffer from false sharing. We verified that these benchmarks do not contain false sharing by running all their available workloads with Huron's detector. We also evaluate Huron on three other benchmarks, `boost_spinlock` (from C++ boost [252] library), `ref_count` (adapted from Java's reference counting implementation [96]), and `histogramFS` (a modified version of `histogram` from the Phoenix [300] suite), which were all previously used by TMI [93]. We note that `boost_spinlock` and `ref_count` are from real world applications.

Aside from these benchmarks, we create and use two additional microbenchmarks, `lockless_writer` and `locked_writer`, that highlight the pros and cons of each false sharing repair technique. Both microbenchmarks incur false sharing due to multiple writer threads writing to the same cache line. As the name implies, `lockless_writer` does not rely on any synchronization instructions, while `locked_writer` uses locks for synchronization between the write operations.

We also use five microbenchmarks with true sharing (*i.e.*, multiple threads accessing overlapping data on the same cache line) when evaluating Huron's accuracy. In particular, state-of-the-art techniques suffer from accuracy issues and can incorrectly detect true sharing instances as false sharing bugs. The `single reader single writer`, `multiple readers single writer`, `multiple readers multiple writers` all read and write truly shared data to/from single/multiple threads. The `atomic writers` concurrently writes data from multiple threads us-

ing C++ atomic primitives [386] and `non-atomic writers` simply performs concurrent writes.
Metrics. Speedups in all the figures are relative to the execution time of the original benchmark.

We report all performance data as an average of 25 runs.

3.5.2 Accuracy of False Sharing Detection

Table 3.1 shows detection results for Huron, TMI and Sheriff. Here, TP stands for true positive, *i.e.*, correctly flagged real false sharing bugs; FP stands for false positive, *i.e.*, a true sharing instance incorrectly flagged as a false sharing bug; FN stands for false negative, *i.e.*, a real false sharing bug that a detector missed; and finally TN stands for true negative, a correct report of non-existence of false sharing.

Table 3.1 also reports the accuracy of each technique as $(TP+TN)/(TP+TN+FP+FN)$. We acknowledge that each technique can incur additional false negatives if the programs were run with different inputs (*e.g.,* besides all the existing test cases and workloads we used). Since such false negatives would impact all the techniques in the same way, we report accuracy numbers based on the executions we observe. In the observed executions, Huron's accuracy is 100%, whereas the accuracy of TMI and Sheriff is 66.67%.

Out of 21 benchmarks, Sheriff and TMI mistakenly detect true sharing as false sharing (*i.e.*, FP) in 4 and 5 benchmarks, respectively. Huron's in-house detection does not incur false positives because of its fine-grained (i.e., cache-level) full memory tracing, as opposed to coarse-grained (e.g., page-level) and sampling-based detection employed by Sheriff or TMI. Huron's in-production false sharing detection can temporarily incur a false positive since it relies on TMI. However, Huron eliminates this false positive for subsequent builds of the program during its in-house detection and repair.

Out of 21 benchmarks, Sheriff and TMI fail to detect false sharing (*i.e.*, FN) in 3 and 2 benchmarks, respectively. Sheriff suffers from false negatives due to reader-writer false sharing, as its detection mechanism compares only writes by different threads within a cache line. TMI's false negatives are due to inaccurate hardware events and sampling.

The detection inaccuracy of Sheriff and TMI has a substantial negative impact on the speedup they provide. We compute speedups for all the cases where Huron correctly detects and fixes a false sharing bug (*i.e.*, TP) and Sheriff and TMI miss (*i.e.*, `histogram`, `boost_spinlock`, `locked_writer` for Sheriff and `reverse_index`, `word_count` for TMI). For these benchmarks, Huron provides up to $5.3\times$ and on average $3.6\times$ greater speedup than Sheriff, and up to $4.3\times$ and on average $2.6\times$ greater speedup than TMI.

Table 3.1: False sharing detection in existing benchmarks.

Benchmark	Sheriff Verdict	TMI Verdict	Huron Verdict
histogram	FN	TP	TP
histogramFS	TP	TP	TP
linear_regression	TP	TP	TP
reverse_index	TP	FN	TP
string_match	TP	TP	TP
lu_ncb	TP	TP	TP
word_count	TP	FN	TP
boost_spinlock	FN	TP	TP
lockless_writer	TP	TP	TP
locked_writer	FN	TP	TP
ref_count	TP	TP	TP
Volrend	TN	TN	TN
radix	TN	TN	TN
ocean	TN	TN	TN
fft	TN	TN	TN
canneal	FP	TN	TN
Single reader single writer	TN	FP	TN
Multiple readers single writer	TN	FP	TN
Multiple readers multiple writers	FP	FP	TN
Atomic writers	FP	FP	TN
Non-atomic writers	FP	FP	TN
Accuracy	**66.67%**	**66.67%**	**100.00%**

3.5.3 Ability to Repair False Sharing Bugs

As shown in Table 3.1, Huron successfully detects and eliminates false sharing in all 11 benchmarks (cells marked True Positive–TP–in Table 3.1). On the other hand, not only Sheriff detects fewer false sharing instances (*i.e.*, 8), it is also only able to repair 5 of them. Sheriff's repair fails for 3 out of 8 cases (`boost_spinlock`, `locked_writer`, `ref_count`) because, to preserve correctness, it is unable to repair false sharing due to synchronization primitives. Similarly, TMI only detects 9 false sharing bugs, out of which it repairs 7. The detection fails in 2 out of the 9 cases (*i.e.*, `lockless_writer`, `locked_writer`), because TMI causes the program to hang.

3.5.4 Effectiveness Comparison to State of the Art

We compare Huron's effectiveness (*i.e.*, the speedup it provides) to the state of the art only for the benchmarks where TMI and Sheriff are able to detect and repair false sharing bugs. These are: `linear_regression`, `histogram`, `histogramFS`, `string_match`, `lu_ncb`, `boost_spinlock`, `lockless_writer`, `locked_writer`, and `ref_count`.

Fig. 3.6 compares the speedup that Huron and Sheriff provide. Huron's speedup outperforms Sheriff's by up to 7.96× and on average 2.72×. Huron performs better than Sheriff for benchmarks with frequent synchronization, where Sheriff's repair mechanism incurs high overhead.

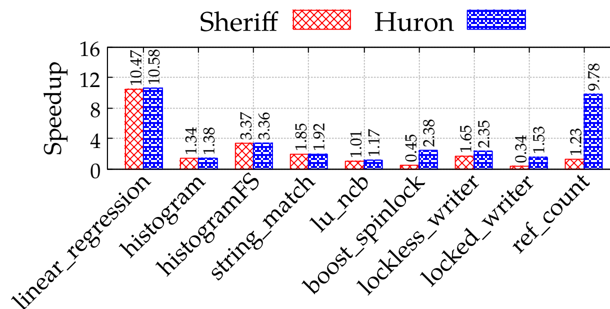


Figure 3.6: Speedup comparison to Sheriff [228]. All standard deviations are less than 1.62%.

Fig. 3.7 compares the speedup that Huron and TMI provide. Huron outperforms TMI by up to 5.7× and on average 2.1×. Similar to Sheriff, TMI's repair is also not as efficient as Huron's for benchmarks with heavy synchronization.Huron is more effective than Sheriff and TMI largely due to its novel in-house repair technique.

Finally, we compare Huron with manual false sharing repair. Interestingly, as shown in Fig. 3.8, Huron outperforms manual repair in 7 out of 9 benchmarks–albeit with a small margin of up to

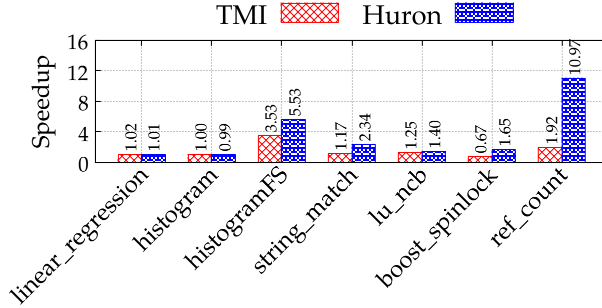


Figure 3.7: Speedup comparison to TMI [93]. All standard deviations are less than 3.2%.

8%. This is because, as explained in §3.3.3, when Huron needs to insert padding, it uses existing program data rather than dummy padding that manual repair uses.

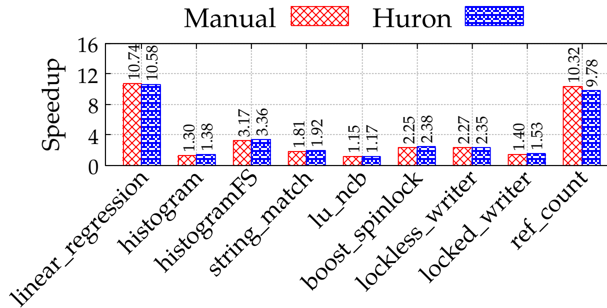


Figure 3.8: Speedup comparison to manual repair. All standard deviations are less than 0.71%.

In summary, compared to the state of the art [228, 93], Huron provides up to 8× (2.11-2.27× average) more speedup.

3.5.5 Effectiveness of Huron's Input-Independent Repair

In this section, we investigate the effectiveness of Huron in generating input-independent repairs based on false sharing bugs detected in house. To illustrate this, we do a detailed analysis of `histogram`, which we have discussed previously in detail as part of the example in Fig. 3.3. The `histogram` benchmark experiences false sharing that is input-dependent. Specifically, the arrays `red` and `blue` (and `green` in the actual program) are involved in input-dependent false sharing.

Fig. 3.9 shows the speedup provided by Huron for `histogram`'s various input images, namely "small", "medium", and "large.". In house, Huron detects false sharing using the "small" input image and generates an input-independent repair that works for the other images in production. For "small", "medium", and "large" input images, the speedup varies between $1.16 - 1.21\times$.

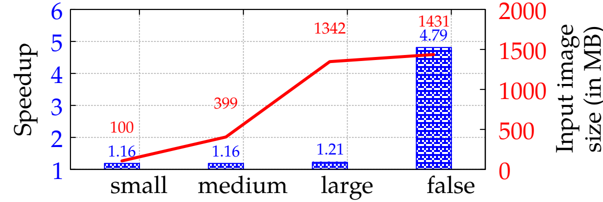


Figure 3.9: Huron' speedup for different input images of `histogram`. Huron generates an input-independent repair for the "small" input. All standard deviations are less than 3.47%.

However, the speedup is $4.79\times$ for the "false" input image. The "false" image (generated from a script provided by the authors of TMI [93]) is actually used as input for `histogramFS`. The pixel values of the image trigger a lot of false sharing and therefore Huron delivers considerable speedup for this benchmark.

3.5.6 Impact of Test Cases on Effectiveness

In this section, we evaluate the impact of test cases on Huron's effectiveness. For this, we initially only rely on Huron's in-production false sharing repair component to simulate a worst case scenario, where Huron does not have access to any test cases in house. Using these results, Huron then repairs all the false sharing instances in house. Fig. 3.10 shows the results. For all the benchmarks, Huron's in-production detection and repair provides speedup that is about the same as TMI's. This is expected since Huron relies on TMI, with the added facility to log metadata that Huron uses to subsequently repair false sharing instances in house. Huron's in-house repair that uses the feedback it receives from its in production component provides greater speedup than TMI.

3.5.7 False Sharing Repair Overhead

We now first evaluate the memory overhead of false sharing repair for Huron, TMI, and Sheriff. We then study the effect of these tools' memory usage on the speedup they provide.

We compare the memory overhead of Huron to the memory overhead of Sheriff and TMI in Fig. 3.11 and Fig. 3.12, respectively. The overheads in both plots are relative to the memory usage of the unmodified binaries. Huron uses up to $377\times$ and on average $59\times$ less memory than Sheriff. We also observe that on average, Huron's memory overhead is less than 8%. Only for `lockless_writer`, Huron incurs a high (60%) memory overhead. This happens because there is not enough data that is being accessed by the same set of threads in this program. Consequently, Huron has to rely on padding to eliminate false sharing, which incurs overhead. Sheriff, on the other hand, uses significantly more memory than the original benchmark, as it transforms each thread into a

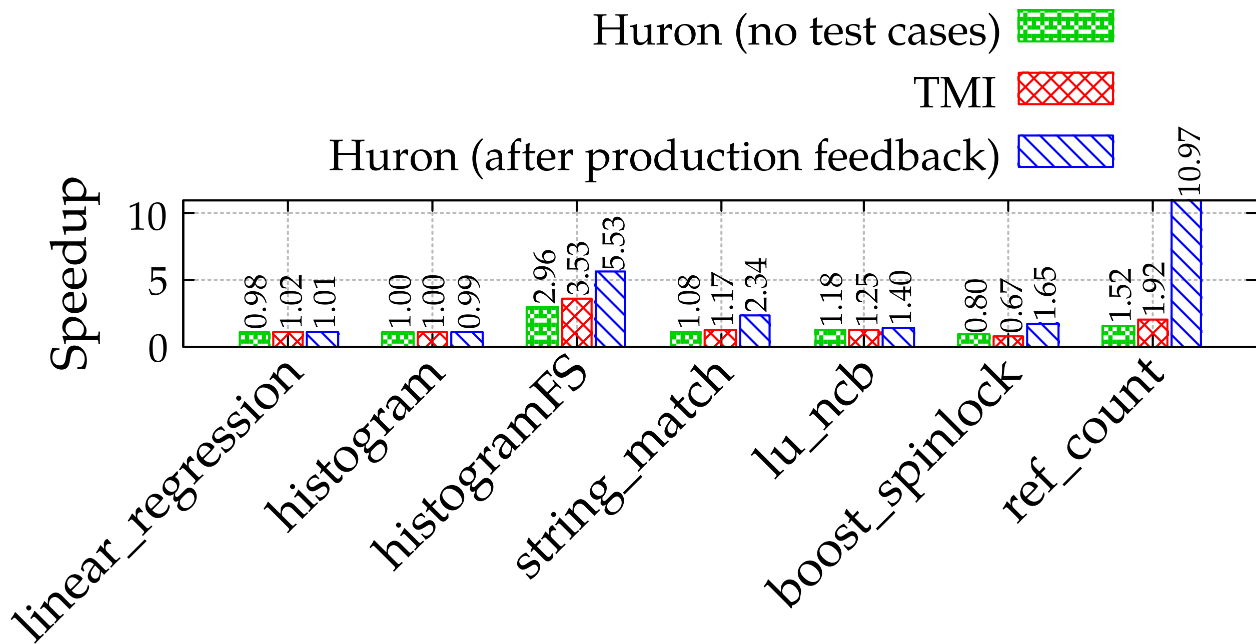


Figure 3.10: Huron's in-house repair that uses the feedback received from its in production component provides greater speedup than TMI.

process (which creates multiple copies of many pages). TMI has a lower memory overhead than Sheriff, thanks to its various optimizations (*e.g.,* thread private memory). However, TMI also uses a few auxiliary buffers to accelerate the detection and elimination of false sharing. Nevertheless, Huron's memory overhead is on average 27× (and up to 197×) lower than TMI. Although Huron's in-production repair leverages TMI, Huron avoids much of TMI's overhead by fixing most false sharing instances in house.

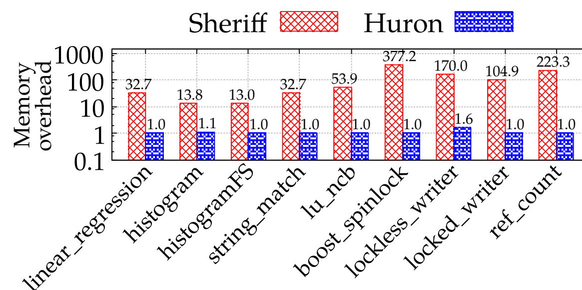


Figure 3.11: Memory overhead comparison to Sheriff. The y-axis is log scale.

The high memory overhead of TMI and Sheriff also has a significant impact on the speedup that they can provide. Specifically, if the underlying system's memory is constrained, a program with a large memory footprint will not enjoy the same speedups that Huron can provide.

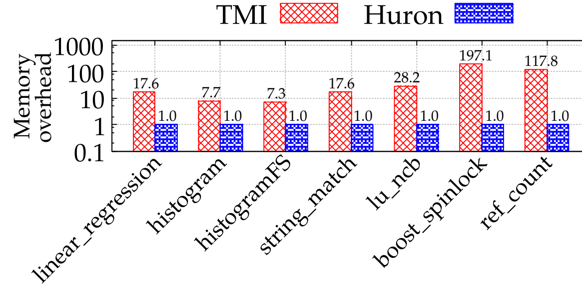


Figure 3.12: Memory overhead comparison to TMI. The y-axis is log scale.

To evaluate the effect of memory pressure, we vary the per-process memory limit from 50 to 25 megabytes and measure the normalized speedup relative to the original benchmark for Sheriff, TMI, and Huron. As shown in Fig. 3.13, Huron provides up to 41% and on average 15% more speedup than Sheriff. Similarly, Huron provides up to 214% and on average 49% more speedup than TMI, as shown in Fig. 3.14.

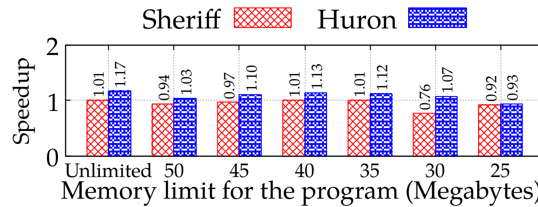


Figure 3.13: Huron achieves up to 41% (15% on average) higher speedup than Sheriff when we limit the memory for the `lu_ncb` benchmark. All standard deviations are less than 0.59%.

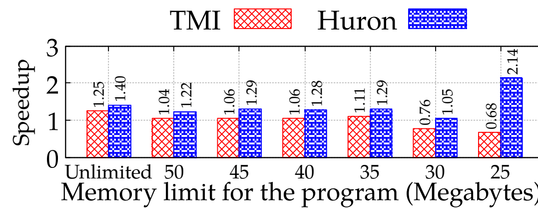


Figure 3.14: Huron achieves up to 214% (49% average) higher speedup than TMI when we limit the memory for the `lu_ncb` benchmark. All standard deviations are less than 4.16%.

3.5.8 Overhead of Huron's In-House Detection

The key advantage of in-house detection is that it is an offline process, and hence does not affect the program execution time in production. However, we still measure the overhead of Huron's

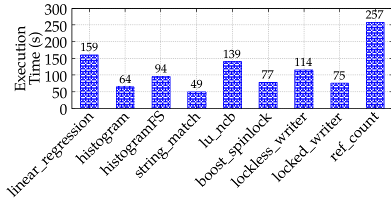


Figure 3.15: Overhead of Huron's in-house detection, in seconds.

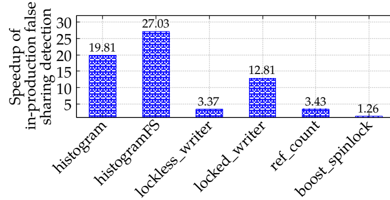


Figure 3.16: Speedup of in-production false sharing detection using Huron's false sharing cache.

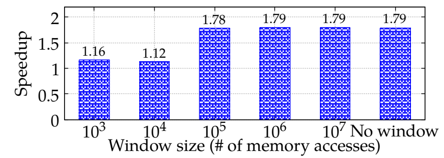


Figure 3.17: The effect of detection window granularity on Huron's speedup (`lockless_writer` benchmark).

in-house detection as the time added to the execution of the user program, *i.e.*, the execution time of the instrumented binary minus that of the unmodified one. In this experiment, we use the same inputs as in §3.5.2. As shown in Fig. 3.15, this overhead is on average less than two minutes (115 seconds) and is no more than 257 seconds. These numbers are quite reasonable for an offline process and on par with the offline overhead of state-of-the-art memory performance profilers [257, 82, 81, 271, 237].

3.5.9 Effect of False Sharing Cache on In-Production Overhead

In this section, we evaluate how the cache of false sharing bugs detected in house reduces the overhead of Huron's in-production false sharing detection. Without Huron's cache, the TMI detector (which Huron relies on) does a lot of extra work to determine (1) the program counter of where false sharing occurs, (2) whether there is a read-write or write-write sharing, (3) whether there is true or false sharing.

As shown in Fig. 3.16, Huron's cache speeds up in-production detection up to $27.1\times$ and on average by $11.3\times$. Note that for 3 out of 9 benchmarks (`linear_regression`, `string_match`, and `lu_ncb`) evaluated in § 3.5.4, speedups are not shown, because TMI removes false sharing from these programs using its allocator even before false sharing instances occur in production (*i.e.*, the cache is never used).

Finally, we measure the impact of caching on memory. In particular, we determine that Huron's cache takes 512KB in the worst case and 91.21KB on average. Considering that the memory overhead of Huron is considerably lower than state-of-the-art tools (on average 27-59×), we consider the modest cache overhead to be acceptable.

3.5.10 Contributions of In-House and In-Production Repair Techniques

We now quantify the extent to which Huron's in-house and in-production repair techniques contribute to the overall efficiency of Huron. For this, we use a benchmark with 10 false sharing

instances. We then vary the number of false sharing instances repaired in house from 1 to 10. Since Huron's in-production repair converts threads into processes, it eliminates all the remaining false sharing bugs at once.

Fig. 3.18 shows our results. The *in-house* speedups are due to Huron's in-house repair only, and the *hybrid* speedups are due to Huron's hybrid in-house/in-production repair. Both in-house and hybrid speedups increase with an increase in the number of false sharing instances repaired in house. The contribution of the in-production component is the difference between the *hybrid* and the *in-house* speedup values. In-house repair contributes more to the overall speedup than in-production repair. For instance, when five of the false sharing bugs are fixed in house with the other five repaired in production, the in-house component provides 39.3% of the speedup, whereas the in-production component provides 6.8% of the speedup. This trend is stable across data points.

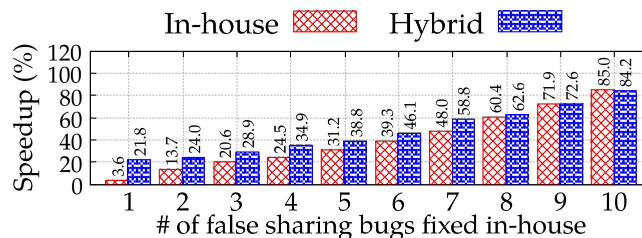


Figure 3.18: Huron's provides greater performance speedup as more false sharing instances are repaired in house. All standard deviations are less than 3.62%.

3.5.11 Effect of Detection Time Window Granularity on Repair Speedup

As discussed in §3.3.2, Huron's false sharing detection time window granularity can impact how many false sharing bugs it detects and fixes, and thus the speedup it provides. In this section, we evaluate the effect of the detection time window granularity on the speedup provided by Huron's repair.

Fig. 3.17 shows the speedup provided by Huron for the `lockless_writer` benchmark under different window granularities. The probability of missing the detection of a false sharing bug due to a large time window increases with the number of falsely-shared cache lines. We choose `lockless_writer`, which we know (via manual inspection of the small code base) suffers from false sharing in exactly 1024 cache lines, the largest number across all benchmarks.

The speedup provided by Huron increases with a larger window size. This is because a small window limits Huron's ability to detect out-of-window memory accesses that are involved in false sharing with in-window memory accesses. Therefore, Huron misses many false sharing instances for smaller window sizes and the speedup after repair is sub-optimal. As the window size increases, so does the number of detected false sharing instances and the ensuing speedup.

3.6 Related Work

There is a substantial amount of related work that has studied the detection and elimination of false sharing. In many cases, existing work attempts to detect and repair false sharing through dynamic analysis [408, 104]. Some approaches also rely on heuristics to make detection and repair more scalable and efficient [246, 71, 79, 125]. Approaches based on static analysis can also detect and eliminate false sharing by reorganizing a program's code [32, 191, 46]. Alas, static false sharing repair [177, 75, 159] was shown to be mostly effective in well-defined use cases [228]. Huron combines the best of both worlds to achieve good accuracy, effectiveness, and efficiency.

Simulators and profilers can be used to detect false sharing. For instance, [312] employs full system simulation using Simics [241] to identify cache miss causes. Other tools [225, 132] detect false sharing by instrumenting memory accesses using Intel Pin [237] or Valgrind [271]. Predator [230] uses LLVM [216] instrumentation to record memory accesses by different threads to detect false sharing. These tools are helpful for detection; however, they provide few hints as to how to best repair. These techniques can also incur high runtime overhead and suffer from false positives [228]. Huron's hybrid approach does not suffer from these problems.

In order to reduce the runtime overhead, many techniques [130, 247, 158, 269, 104, 239, 146, 244, 93, 64, 285, 229] rely on performance counter values that are correlated with false sharing (i.e, Intel HITM [83]). Once the counter events surge beyond a certain threshold, these techniques trigger a more rigorous detection algorithm [104, 239, 93]. Huron uses a similar approach for its in-production false sharing detection. However, because Huron can detect and repair many false sharing instances in house, it does not trigger in-production false sharing detection frequently, and thus incurs low overhead.

Another technique uses machine learning on hardware event counts [158] to detect false sharing. We plan to improve Huron by leveraging a machine learning-based approach to speed up its in-house false sharing detection.

A common false sharing elimination approach in prior work is to transform program threads to processes so that they no longer share the same address space. While Grace [49] first proposed this idea to avoid concurrency bugs, Sheriff [228] adopted this technique to repair false sharing. This approach incurs high memory overheads and can be inefficient because the pages shared across processes need to be merged frequently. TMI [93] partially addresses these challenges by introducing a data structure called page twinning store buffer (PTSB). PTSB has smaller memory footprint, and it allows pages to be merged more efficiently. Despite these benefits, the speedup provided by PTSB (and thus TMI) cannot fully attain the performance benefits provided by manual repair or Huron (see §3.5.4). Finally, unlike Sheriff and TMI, Huron does not suffer from false positives.

Many other studies [394, 173, 176, 175, 174, 178, 307, 171, 331, 123, 406, 272, 98] have investigated data layout optimizations to improve performance. The main goal of these tools is to improve the memory layout to maximize spatial locality, while Huron utilizes memory layout transformations to eliminate false sharing.

3.7 Conclusion

Detecting and fixing all false sharing is difficult. Even if false sharing instances are identified during development, repairing them manually can be a daunting task. In this paper, we described Huron, a hybrid in-house/in-production mechanism that detects and repairs false sharing automatically. Huron's repair mechanism groups together data accessed by the same set of threads to shift falsely-shared data to different cache lines. Huron detects and repairs all false sharing bugs with 100% accuracy in the 21 benchmarks that we evaluated. Huron achieves speedups of up to 11$\times$ and on average 3.82$\times$. Overall, Huron is 33.33% more accurate than state-of-the-art detection and repair tools and it provides up to 8$\times$ and on average 2.11-2.27$\times$ greater speedup.

CHAPTER 4

I-SPY: Context-Driven Conditional Instruction Prefetching with Coalescing

Modern data center applications have rapidly expanding instruction footprints that lead to frequent instruction cache misses, increasing cost and degrading data center performance and energy efficiency. Mitigating instruction cache misses is challenging since existing techniques (1) require significant hardware modifications, (2) expect impractical on-chip storage, or (3) prefetch instructions based on inaccurate understanding of program miss behavior.

To overcome these limitations, we[1] first investigate the challenges of effective instruction prefetching. We then use insights derived from our investigation to develop I-SPY, a novel profile-driven prefetching technique. I-SPY uses dynamic miss profiles to drive an offline analysis of I-cache miss behavior, which it uses to inform prefetching decisions. Two key techniques underlie I-SPY's design: (1) *conditional prefetching*, which only prefetches instructions if the program context is known to lead to misses, and (2) *prefetch coalescing*, which merges multiple prefetches of non-contiguous cache lines into a single prefetch instruction. I-SPY exposes these techniques via a family of light-weight hardware code prefetch instructions.

We study I-SPY in the context of nine data center applications and show that it provides an average of 15.5% (up to 45.9%) speedup and 95.9% (and up to 98.4%) reduction in instruction cache misses, outperforming the state-of-the-art prefetching technique by 22.5%. We show that I-SPY achieves performance improvements that are on average 90.5% of the performance of an ideal cache with no misses.

[1]Some of the work in this chapter was performed in collaboration with Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci [198]. Therefore, I use the "we" pronoun in this chapter to acknowledge their involvement in this work.

4.1 Introduction

The expanding user base and feature portfolio of modern data center applications is driving a precipitous growth in their complexity [179]. Data center applications are increasingly composed of deep and complex software stacks with several layers of kernel networking and storage modules, data retrieval, processing elements, and logging components [41, 211, 212]. As a result, code footprints are often a hundred times larger than a typical L1 instruction cache (I-cache) [39], and further increase rapidly every year [179].

I-cache misses are becoming a critical performance bottleneck due to increasing instruction footprints [112, 179, 41]. Even modern out-of-order mechanisms do not hide instruction misses that show up as glaring stalls in the critical path of execution. Hence, reducing I-cache misses can significantly improve data center application performance, leading to millions of dollars in cost and energy savings [344, 41].

The importance of mechanisms that reduce I-cache misses (e.g., instruction prefetching) has long been recognized. Prior works have proposed next-line or history-based hardware instruction prefetchers [328, 302, 114, 280, 113, 189, 209, 212, 211] and several software mechanisms have been proposed to perform code layout optimizations for improving instruction locality [236, 221, 65, 276, 278]. While these techniques are promising, they (1) demand impractical on-chip storage [113, 114, 189], (2) require significant hardware modifications [212, 211], or (3) face inaccuracies due to approximations used in computing a cache-optimal code layout [287, 276].

A recent profile-guided prefetching proposal, AsmDB [41], was able to reduce I-cache misses in Google workloads. However, we find that even AsmDB can fall short of an ideal prefetcher by 25.5% on average. To completely eliminate I-cache misses, it is important to first understand: why do existing state-of-the-art prefetching mechanisms achieve sub-optimal performance? What are the challenges in building a prefetcher that achieves near-ideal application speedup?

To this end, we perform a comprehensive characterization of the challenges in developing an ideal instruction prefetcher. We find that an ideal instruction prefetcher must make careful decisions about (1) *what* information is needed to efficiently predict an I-cache miss, (2) *when* to prefetch an instruction, (3) *where* to introduce a prefetch operation in the application code, and (4) *how* to sparingly prefetch instructions. Each of these design decisions introduces non-trivial trade-offs affecting performance and increasing the burden of developing an ideal prefetcher. For example, the state-of-the-art prefetcher, AsmDB, injects prefetches at link time based on application's miss profiles. However, control flow may not be predicted at link time or may diverge from the profile at run time (e.g., due to input dependencies), resulting in many prefetched cache lines that never get used and pollute the cache. Moreover, AsmDB suffers from static and dynamic code bloat due to additional prefetch instructions injected into the code.

In this work, we aim to reduce I-cache misses with I-SPY—a prefetching technique that carefully identifies I-cache misses, sparingly injects "code prefetch" instructions in suitable program locations at link time, and selectively executes injected prefetch instructions at run time. I-SPY proposes two novel mechanisms that enable on average 90.4% of ideal speedup: *conditional prefetching* and *prefetch coalescing*.

Conditional prefetching. Prior techniques [238, 41] either prefetch excessively to hide more I-cache misses, or prefetch conservatively to prevent unnecessary prefetch operations that pollute the I-cache. To hide more I-cache misses as well as to reduce unnecessary prefetches, we propose *conditional prefetching*, wherein we use profiled execution context to inject code prefetch instructions that cover each miss, at link time. At run-time, we reduce unnecessary prefetches by executing an injected prefetch instruction only when the miss-inducing context is observed again.

To implement conditional prefetching with I-SPY, we propose two new hardware modifications. First, we propose simple CPU modifications that use Intel's Last Branch Record (LBR) [7] to enable a server to selectively execute an injected prefetch instruction based on the likelihood of the prefetch being successful. We also propose a "code prefetch" instruction called `Cprefetch` that holds miss-inducing context information in its operands, to enable an I-SPY-aware CPU to conditionally execute the prefetch instruction.

Prefetch coalescing. Whereas conditional prefetching facilitates eliminating more I-cache misses without prefetching unnecessarily at run time, it can still inject too many prefetch instructions that might further increase the static code footprint. Since data center applications face significant I-cache misses [179, 344], injecting even a single prefetch instruction for each I-cache miss can significantly increase an already-large static code footprint. To avoid a significant code footprint increase, we propose *prefetch coalescing*, wherein we prefetch multiple cache lines with a single instruction. We find that several applications face I-cache misses from non-contiguous cache lines, i.e., in a window of N lines after a miss, only a subset of the N lines will incur a miss. We propose a new instruction called `Lprefetch` to prefetch these non-contiguous cache lines using a single instruction.

We study I-SPY in the context of nine popular data center applications that face frequent I-cache misses. Across all applications, we demonstrate an average performance improvement of 15.5% (up to 45.9%) due to a mean 95.9% (up to 98.4%) L1 I-cache miss reduction. We also show that I-SPY improves application performance by 22.4% compared to the state-of-the-art instruction prefetcher [41]. I-SPY increases the dynamically-executed instruction count by 5.1% on average and incurs an 8.2% mean static code footprint increase.

In summary, we make the following contributions:

- A detailed analysis of the challenges involved in building a prefetcher that provides close-to-ideal speedups.

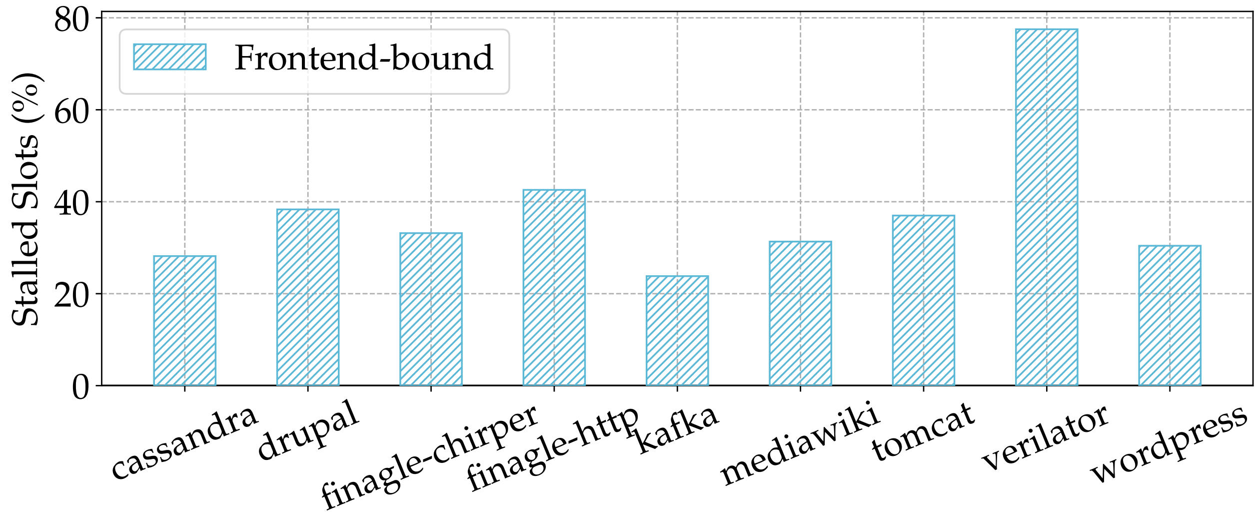


Figure 4.1: Several widely-used data center applications spend a significant fraction of their pipeline slots on "Frontend-bound" stalls, waiting for I-cache misses to return (measured using the Top-down methodology [395]).

- *Conditional prefetching:* A novel profile-guided prefetching technique that accurately identifies miss-inducing program contexts to prefetch I-cache lines only when needed.
- *Prefetch coalescing:* A technique that coalesces multiple non-contiguous cache line prefetches based on run-time information obtained from execution profiles.
- I-SPY: An end-to-end system that combines conditional prefetching with prefetch coalescing using a new family of instructions to achieve near-ideal speedup.

4.2 Understanding the Challenges of Instruction Prefetching

In this section, we present a detailed characterization of the challenges in developing an ideal instruction prefetching technique. We define an ideal prefetcher as one that achieves the performance of an I-cache with no misses, i.e., where every access hits in the L1 I-cache (a theoretical upper bound). We characterize prefetching challenges by exploring four important questions: (1) ***What*** information is needed to efficiently predict an I-cache miss?, (2) ***When*** must an instruction be prefetched to avoid an I-cache miss? (3) ***Where*** should a prefetcher inject a code prefetch instruction in the program?, and (4) ***How*** can a prefetcher sparingly prefetch instructions while still eliminating most I-cache misses?

We characterize challenges using nine popular real-world applications that exhibit significant I-cache misses. In Fig. 4.1, we show the "frontend" pipeline stalls that the nine applications exhibit when waiting for I-cache misses to return. We observe that these data center applications can spend 23% - 80% of their pipeline slots in waiting for I-cache misses to return. Hence, we include these

applications in our study.

From Facebook's HHVM OSS-performance [19] benchmark suite, we analyze (1) *Drupal*: a PHP content management system, (2) *Mediawiki*: an open-source Wiki engine, and (3) *Wordpress*: a PHP-based content management system used by services such as Bloomberg Professional and Microsoft News. From the Java DaCapo [51] benchmark suite, we analyze (a) *Cassandra* [2]: a NoSQL database management system used by companies such as Instagram and Netflix, (b) *Kafka*: Apache's stream-processing software platform used by companies such as Uber and Linkedin, and (c) *Tomcat* [4]: Apache's implementation of the Java Servlet and WebSocket. From the Java Renaissance [294] benchmark suite, we analyze *Finagle-Chirper* and *Finagle-HTTP* [15]: Twitter Finagle's micro-blogging service and HTTP server, respectively. We also study *Verilator* [16], a tool used by cloud companies to simulate custom hardware designs. We describe our complete experimental setup and simulation parameters in Sec. 4.5.

4.2.1 What Information is Needed to Efficiently Predict an I-Cache Miss?

An ideal prefetcher must predict all I-cache misses before they occur, to prefetch them into the I-cache in time. To this end, prior work [328, 302, 114, 41] (e.g., next-in-line prefetching) has shown that an I-cache miss can be predicted using the program instructions executed before the miss. Since any arbitrary instruction (e.g., direct/indirect branches or function returns) could execute before a miss, the application's dynamic control flow must be tracked to predict a miss using the program paths that lead to it. An application's execution can be represented by a dynamic Control Flow Graph (CFG). In a dynamic CFG, the nodes represent basic blocks (sequence of instructions without a branch) and the edges represent branches. Fig. 4.2 shows a dynamic CFG, where the cache miss at basic block K can be reached via various paths. The CFG's edges are typically weighted by a branch's execution count. For brevity, we assume all the weights are equal to one in this example.

Software-driven prefetchers [65, 278, 41] construct an application's dynamic CFG and identify miss locations that can be eliminated using a suitable prefetch instruction. For example, AsmDB [41] uses DynamoRIO's [58] memory trace client to capture an application's dynamic CFG for locating I-cache misses in the captured trace. Unfortunately, DynamoRIO [58] incurs undue overhead [138], making it costly to deploy in production. To efficiently generate miss-annotated dynamic CFGs, we propose augmenting dynamic CFG traces from Intel's LBR [7] with L1 I-cache miss profiles collected with Intel's Precise Event Based Sampling (PEBS) [99] performance counters. Generating dynamic CFGs using such lightweight monitoring enables profiling applications in production.

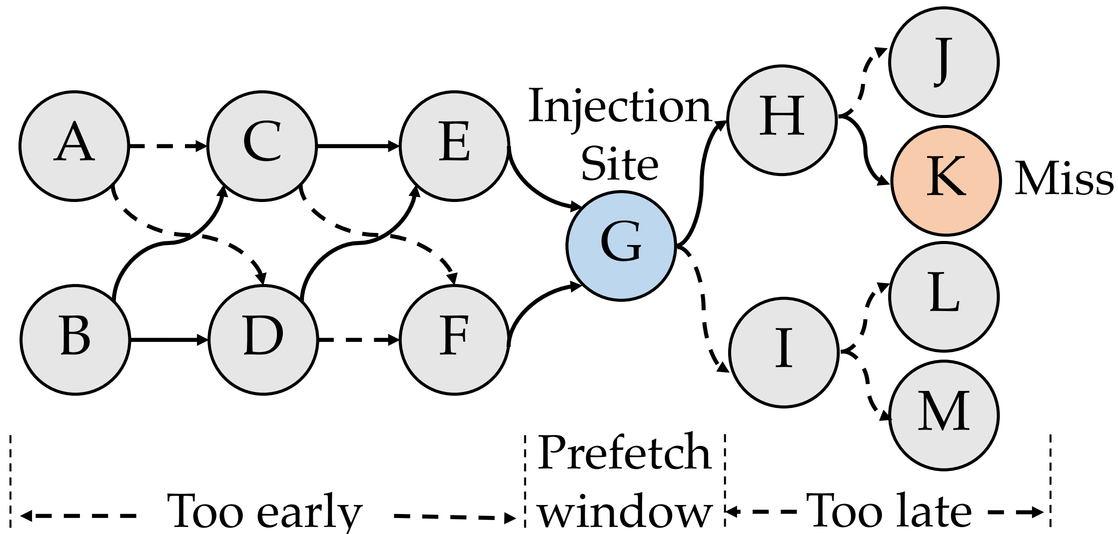


Figure 4.2: A partial example of a miss-annotated dynamic control flow graph. Dashed edges represent execution paths that do not lead to a miss.

Observation: Representing a program's execution using a dynamic CFG and augmenting it with L1 I-cache miss profiles enables determining prefetch candidates.
Insight: *Generating a lightweight miss-annotated dynamic CFG using Intel's LBR and PEBS incurs low run-time performance overhead and enables predicting miss locations in production systems.*

4.2.2 When To Prefetch an Instruction?

A prefetch is successful only if it is timely. In the dynamic CFG in Fig. 4.2, a prefetch instruction injected at predecessor basic blocks H or I is too late: the prefetcher will not be able to bring the line into the I-cache in time and a miss will occur at K. In contrast, if a prefetch instruction is injected at predecessors E or F, the prefetched line may not be needed soon enough, and it may (1) either evict other lines that will be accessed sooner, or (2) itself get prematurely evicted before it is accessed. Instead, the prefetch must be injected in an appropriate *prefetch window*. In our example, we assume block G is a timely injection candidate in the prefetch window.

Prior work [41] empirically determines an ideal prefetch window using average application-specific IPC to inject a prefetch instruction that hides a cache miss. I-SPY relies on this approach and injects prefetch instructions 27 - 200 cycles before a miss, a window we determine in our evaluation.

Observation: An instruction must be prefetched in a timely manner to avoid a miss.
Insight: *Empirically determining the prefetch window such that a prefetch is not too early or too late, can effectively eliminate a miss.*

4.2.3 Where to Inject a Prefetch?

An ideal prefetcher would eliminate all I-cache misses, achieving full *miss coverage*. To achieve full miss coverage, a prefetcher such as the one proposed by Luk and Mowry [238], might inject a "code prefetch" instruction into every basic block preceding an I-cache miss. However, the problem of this approach is that due to dynamic control flow changes, naively injecting a prefetch into a predecessor basic block causes a high number of inaccurate prefetches whenever the predecessor does not lead to the miss. Prefetching irrelevant lines hurts *prefetch accuracy* (the fraction of useful prefetches) and leads to I-cache pollution, degrading application performance.

Prefetch accuracy can be improved by assessing the usefulness of a prefetch and by restricting the injection of prefetches to those that are likely to improve performance. To determine the likelihood of a prefetch being useful, we can analyze the *fan-out* of the prefetch's injection site. We define fan-out as the percentage of paths that do not lead to a target miss from a given injection site. For example, in Fig. 4.2, the candidate injection site G has a fan-out of 75% as only one out of four paths leads to the miss K.

By limiting prefetch injection to nodes whose fan-out is below a certain threshold, accuracy can be improved, however, coverage is also reduced. The fan-out threshold that decides whether to inject a prefetch represents a control knob to trade-off coverage vs. accuracy. To determine this threshold, Fig. 4.3 analyzes the impact of fan-out on accuracy and coverage for the *wordpress* application. As it can be seen, for real applications with large CFGs, a high fan-out of 99% is required to achieve the best performance, although accuracy starts to drop sharply at this point. Hence, prior works (including AsmDB) that rely on static analysis for injecting prefetches fall short of achieving close to ideal performance (65% in the case of *wordpress*).

With I-SPY, we aim to break this trade-off by optimizing for prefetch accuracy and miss coverage simultaneously. To this end, we propose context sensitive conditional prefetching, a technique that statically injects prefetches to cover each miss (i.e., high miss coverage), but dynamically executes injected prefetches only when the prefetch is likely to be successful, minimizing unused prefetches (i.e., high prefetch accuracy). In Section 4.3.1, we describe our conditional prefetching technique and our approach that leverages dynamic context information to decide whether to execute a prefetch or not.

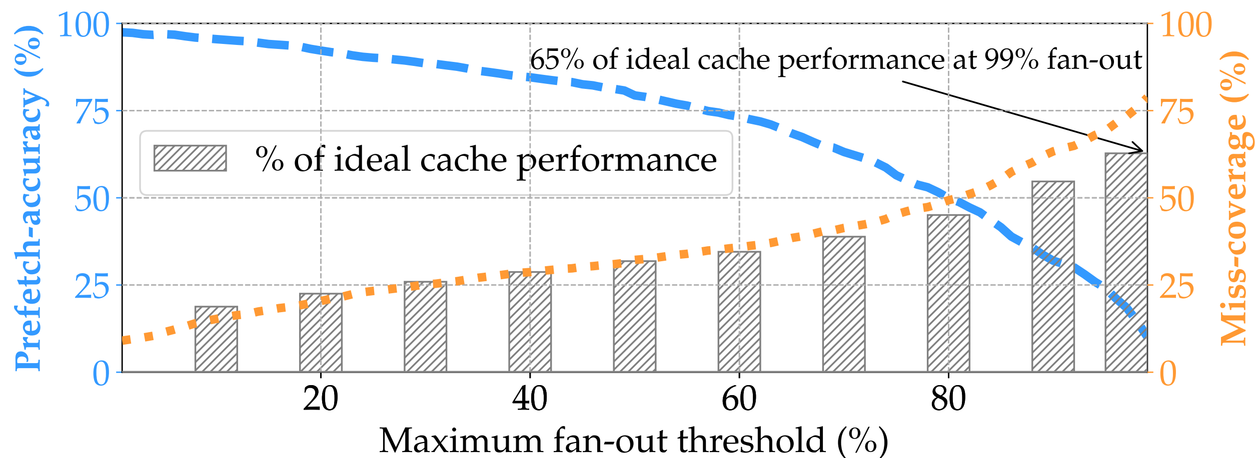


Figure 4.3: Prefetch accuracy vs. miss coverage tradeoff in AsmDB and its relation to ideal cache performance: Miss-coverage increases with an increase in fan-out threshold, but prefetch accuracy starts to reduce. Only 65% of ideal cache performance can be reached at 99% fan-out due to low prefetch accuracy.

Observation: It is challenging to achieve both high miss coverage and prefetch accuracy if we determine prefetch injection candidate blocks based on a static CFG analysis alone.
Insight: *Leveraging dynamic run-time information to conditionally execute statically-injected prefetch instructions can help improve both miss coverage and prefetch accuracy.*

4.2.4 How to Sparingly Prefetch Instructions?

Several profile-guided prefetchers [41, 238] require at least one code prefetch instruction to mitigate an I-cache miss. For example, the state-of-the-art prefetcher, AsmDB [41], covers each miss by injecting a prefetch instruction into a high fan-out (≤99%) predecessor. However, statically injecting numerous prefetch instructions and executing them at run time, increases the static and dynamic application code footprint by 13.7% and 7.3% respectively, as portrayed in Fig. 4.4. An increase in static and dynamic code footprints can pollute the I-cache and cause unnecessary cache line evictions, further degrading application performance. Hence, it is critical to sparingly prefetch instructions to minimize code footprints.

Prefetch coalescing. Our conditional prefetching proposal allows statically injecting more prefetch instructions to eliminate more I-cache misses, without having to dynamically perform inaccurate prefetches. However, a large number of statically-injected code prefetch instructions can still increase an application's static code footprint.

A naïve approach to statically inject fewer instructions is to leverage the spatial locality of I-cache misses to prefetch multiple contiguous cache lines with a single prefetch instruction rather

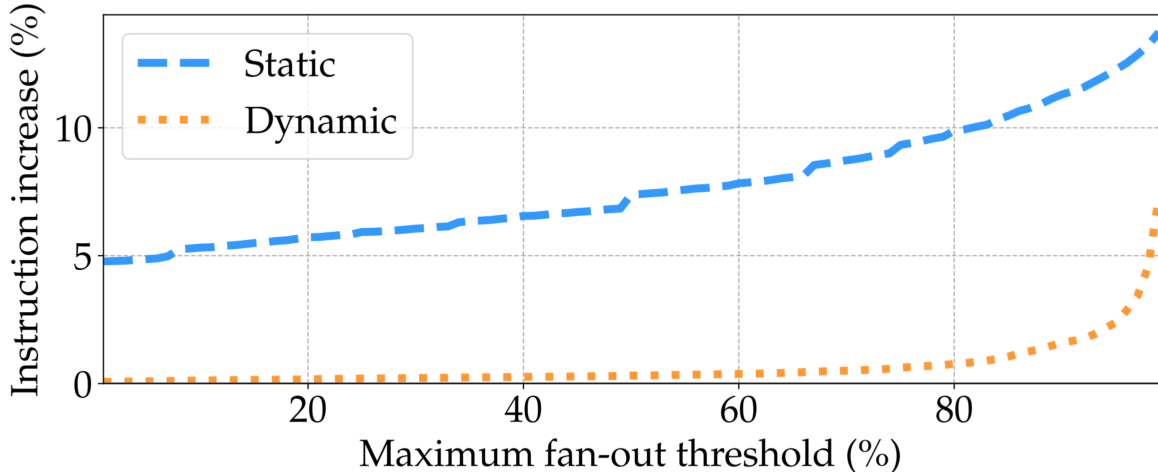


Figure 4.4: AsmDB's static and dynamic code footprint increase: Injecting prefetches in high fan-out predecessors significantly increases static and dynamic code footprints.

than a single line at a time [328, 302]. In contrast, another approach [211] finds value in prefetching multiple non-contiguous cache lines together. Similarly, we posit that it is unlikely that all the contiguous cache lines in a window of n lines after a given miss will incur misses. It is more likely that a subset of the next-n lines will incur misses, whereas others will not. To validate this hypothesis, we consider a window of eight cache lines immediately following a miss to implement two prefetchers: (1) *Contiguous-8*, that prefetches all eight contiguous cache lines after a miss and (2) *Non-contiguous-8*, that prefetches only the missed cache lines in the eight cache line window.

We profile all our benchmarks to detect I-cache misses and measure the speedup achieved by both prefetchers in Fig. 4.5. We find that Non-contiguous-8 provides an average 7.6% speedup over Contiguous-8. We conclude that prefetch coalescing of non-contiguous, but spatially nearby I-cache misses, via a single prefetch instruction can improve performance while minimizing the number of static and dynamic prefetch instructions. We note that our conclusion holds for larger windows of cache lines (e.g., 16 and 32). We find that a window of eight lines offers a good trade-off between speedup and circuit complexity required to support a larger window size. We provide a sensitivity analysis for window sizes in §4.6.2.

Observation: Injecting too many prefetch instructions can increase static and dynamic code footprints, inducing additional cache line evictions.
Insight: *Conditional prefetching can minimize dynamic code footprints; coalescing spatially-near non-contiguous I-cache miss lines into a single prefetch instruction can minimize both static and dynamic code footprints.*

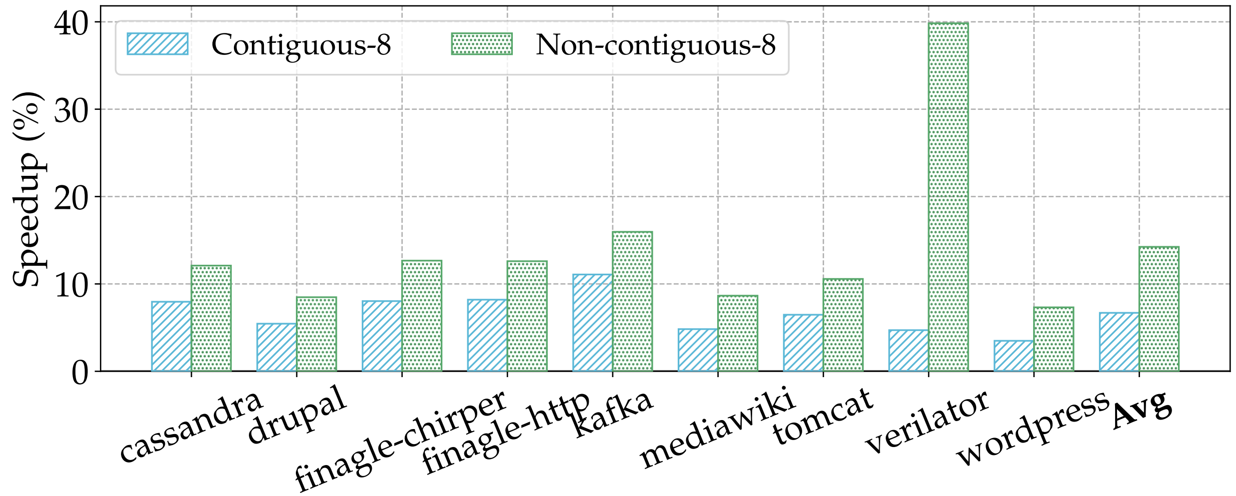


Figure 4.5: Speedup of Contiguous-8 (prefetches all 8 contiguous lines after a miss) vs. Non-contiguous-8 (prefetches only the misses in an 8-line window after a miss): Prefetching non-contiguous cache lines offers a greater speedup opportunity.

4.3 I-SPY

I-SPY proposes two novel techniques to improve profile-guided instruction prefetching. I-SPY introduces *conditional prefetching* to address the dichotomy between high coverage and accuracy discussed in §4.2.3. Furthermore, I-SPY proposes *prefetch coalescing* to reduce the static code footprint increase due to injected prefetch instructions explored in §4.2.4. I-SPY relies on profile-guided analysis at link-time to determine frequently missing blocks and prefetch injection sites using Intel LBR [7] and PEBS [99]. We provide a detailed description of I-SPY's usage model in §4.4. I-SPY also introduces minor hardware modifications to improve prefetch efficiency at run time. As a result, our proposed techniques close the gap between static and dynamic prefetching by combining the performance of dynamic hardware-based mechanisms with the low complexity of static software prefetching schemes.

4.3.1 Conditional Prefetching

Conditionally executing prefetches has a two-fold benefit: I-SPY can liberally inject conditional prefetch instructions to cover each miss (i.e., achieve high miss coverage) while simultaneously minimizing unused prefetches (i.e., achieve high accuracy). I-SPY uses the execution context to decide whether to conditionally execute a prefetch or not. We first discuss how I-SPY computes contexts leading to misses. We then explain how I-SPY 's conditional prefetching instruction is implemented, and finally discuss micro-architectural details.

Miss context discovery. Similar to many other branch prediction schemes [53, 60, 212, 211], I-SPY uses the basic block execution history to compute the execution context. Initially, we attempted to use the exact basic block sequence to predict a miss. However, we found this approach intractable since the number of block sequences (i.e., the number of execution paths) leading to a miss grows exponentially with the increase in sequence length. As a result, I-SPY only considers the presence of certain important basic blocks in the recent context history to inform its prefetching decisions. This approach is in line with prior work [326] that observes that prediction accuracy is largely insensitive to the basic block order sequence.

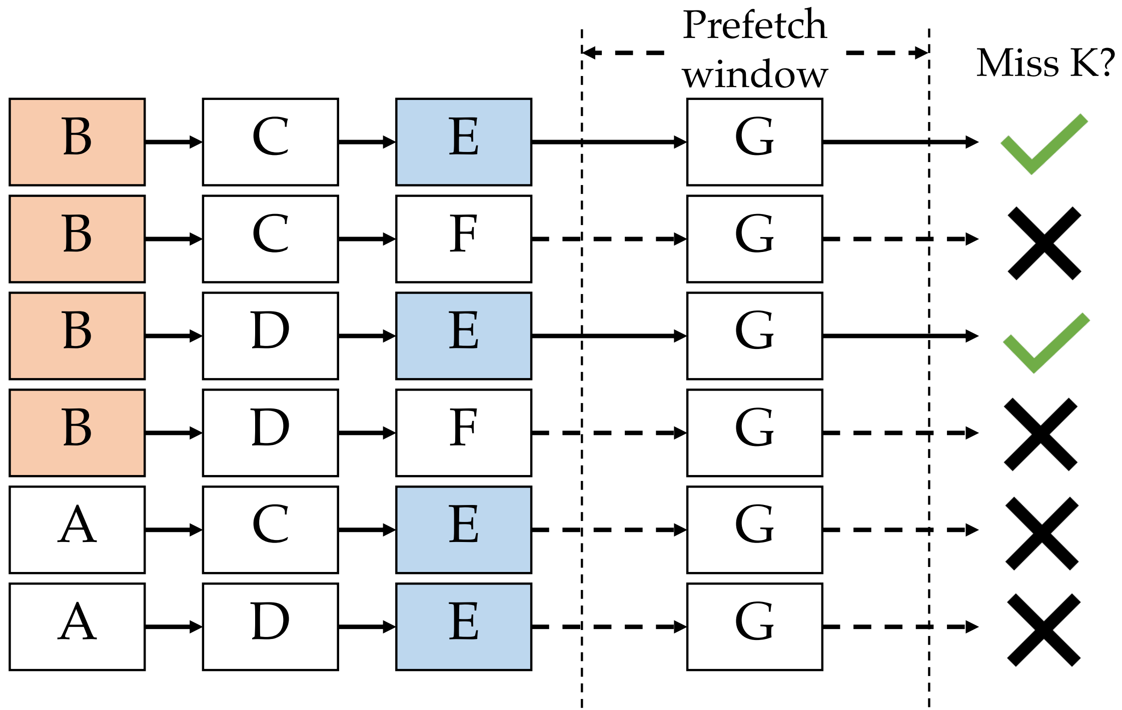
We use the dynamic CFG in Fig. 4.2 to describe the miss context discovery process. Recall that in this example, the miss occurs in basic block K and block G is the injection site in the prefetch window. As shown in Fig. 4.6a, there are six execution paths including the candidate injection site G and two of these paths lead to the basic block K, where the miss occurs.

I-SPY starts miss context discovery by identifying *predictor basic blocks*—blocks with the highest frequency of occurrence in the execution paths leading to each miss. In our example, B and E are predictor blocks. Since I-SPY only relies on the presence of blocks to identify the context (as opposed to relying on the order of blocks), it computes combinations of predictor blocks as potential $contexts$ for a given miss. Then, I-SPY calculates the conditional probability of each $context$ leading to a miss in a block B, i.e., $P(\text{Miss } in\ Block\ \text{“}B\text{”} | \text{context})$ as per the Bayes theorem. As shown in Fig. 4.6b, I-SPY computes $P(\text{Miss } K|\text{B})$, $P(\text{Miss } K|\text{E})$, and $P(\text{Miss } K|\text{B} \cap \text{E})$, i.e., the probability of leading to the miss in block K, given an execution context of either (B), or (E), or both (B and E).

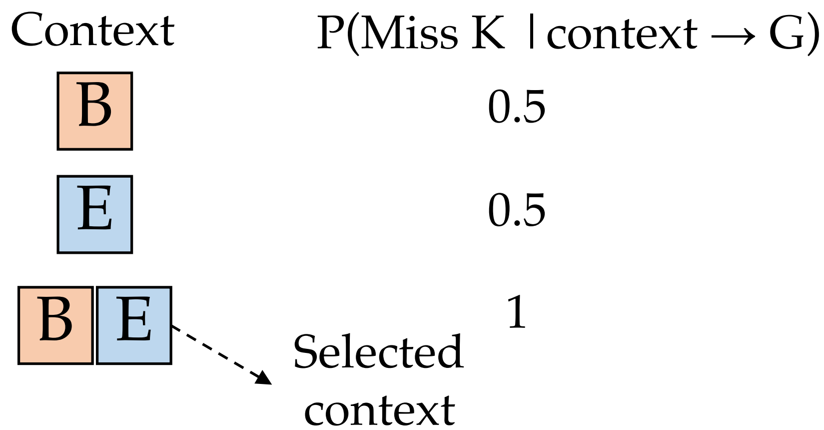
I-SPY then selects the combination with the highest probability as the context for a given miss. In our example, this context, namely (B and E) will be encoded into the conditional prefetch instruction injected at G. At run time, the conditional prefetch will be executed if the run-time branch history contains the recorded context. We now detail I-SPY's conditional prefetch instruction.

Conditional prefetch instruction. We propose a new prefetch instruction, `Cprefetch` that requires an extra operand to specify the execution context. Each basic block in the context is identified by its address, i.e., the address of the first instruction in the basic block. I-SPY computes the basic block address using the LBR data.

To reduce the code size of `Cprefetch`, I-SPY hashes the individual basic block addresses in the context into an n-byte `immediate` operand (`context-hash`) using hash functions, FNV-1 [373] and MurmurHash3 [381]. When a `Cprefetch` is executed at run time, the processor recomputes a hash value (`runtime-hash`) using the last 32 predecessor basic blocks (Intel LBR [7] provides the addresses of 32 most recently executed basic blocks), and compares it against the `context-hash`. The prefetch operation is performed only if the set-bits in `context-hash` are a subset of the set-bits in the `runtime-hash`.



(a) All executions of basic block G including executions that lead to a miss



(b) Probability calculation for a context leading to a miss

Figure 4.6: An example of I-SPY's context discovery process

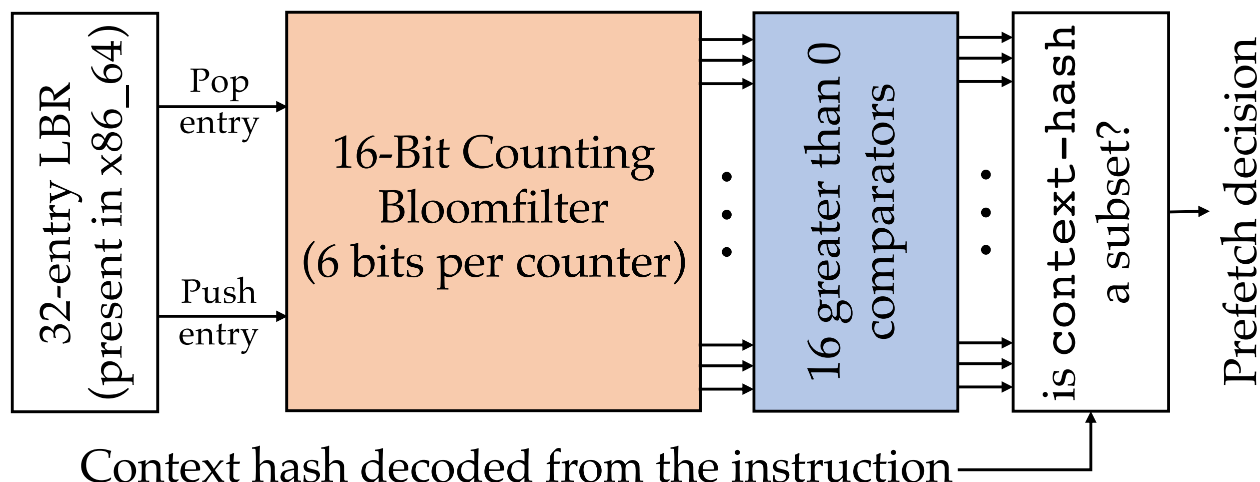


Figure 4.7: Micro-architectural changes needed to execute the context-sensitive conditional prefetch instruction, `Cprefetch`

Both `runtime-hash` and `context-hash` are compressed representations of multiple basic block addresses. While compressing multiple 64 bit basic block addresses into fewer bits reduces the code bloat, it might also introduce false positives. A false positive might occur when the set-bits in `context-hash` are a subset of the set-bits in `runtime-hash`, however, not all the basic blocks represented by `context-hash` are present among the 32 most recently-executed basic blocks represented by `runtime-hash`. We analyze a range of values for the `context-hash` size in Fig. 4.21 and determine that a 16 bit immediate offers a good tradeoff between code bloat and false positive rates.

Micro-architectural modifications `Cprefetch` requires minor micro-architectural modifications. Intel's Xeon data center processors support an LBR [7] control flow tracing facility, which tracks the program counter and target address of the 32 most recently executed branches.

I-SPY extends the LBR to maintain a rolling `runtime-hash` of its contents. Fig. 4.7 shows the micro-architectural requirements of I-SPY's context-sensitive prefetch instruction for 32 predecessor basic blocks and a 16 bit `context-hash`. Since the LBR is a FIFO, we maintain the `runtime-hash` incrementally. Using a counting Bloom filter [110, 56], we assign a 6-bit counter to each of the 16 bits of the `runtime-hash` (96 bits in total). Whenever a new entry is added into the LBR, we hash the corresponding block address and increment the corresponding counters in the `runtime-hash`; the counters for the hash of the evicted LBR entry are decremented. The counters never overflow and the `runtime-hash` precisely tracks the LBR contents since there are only ever 32 branches recorded in the `runtime-hash`. We also add a small amount of logic to reduce each counter to a single "is-zero" bit; in those 16 bits, we check if the `context-hash` bits are a subset of the `runtime-hash`. If they are, the prefetch fires, otherwise it is disabled.

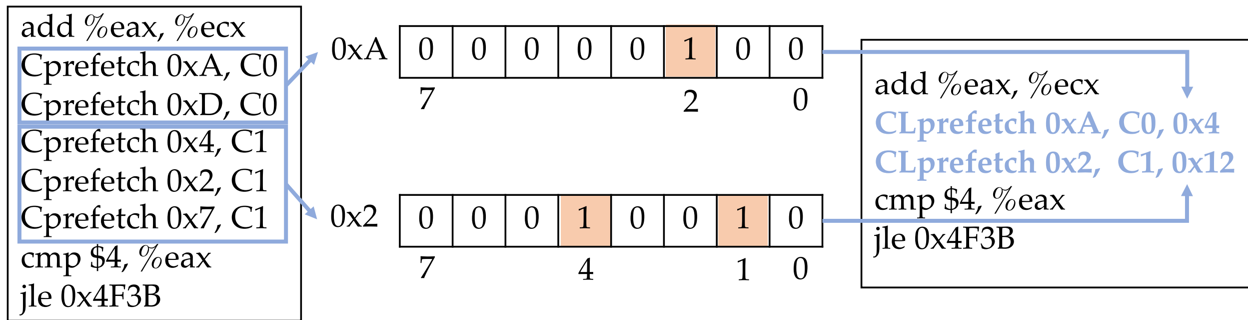


Figure 4.8: An example of I-SPY's prefetch coalescing process

To clarify how Bloom filters help I-SPY match `runtime-hash` to `context-hash`, let's consider the same example in Fig. 4.6. Let's assume the 16-bit hashes of B and E are `0x2` and `0x10`, respectively. Therefore, the `context-hash` would be `0x12`, where the Least Significant Bits (LSB) 1 and 4 are set. To enable prefetching, `runtime-hash` must also have these bits set. At run time, if B is present in the last 32 predecessors, the bloom filter counter corresponding to LSB-1 must be greater than 0. Similarly for E, the counter corresponding to LSB-4 must be greater than 0. Hence, the result of subset comparison between `context-hash` and `runtime-hash` will be true and a prefetch will be triggered.

4.3.2 Prefetching Coalescing

Conditional prefetching enables high-accuracy prefetching. Nevertheless, it leads to static code bloat as every prefetch instruction increases the size of the application's text segment. *Prefetch coalescing* reduces the static code bloat as well as the number of dynamically-executed prefetch instructions by combining multiple prefetches into a single instruction. We first describe how I-SPY decides which lines should be coalesced, followed by details of I-SPY's coalesced prefetching instruction. We then detail the micro-architectural modifications required to support prefetch coalescing.

To perform coalescing, I-SPY analyzes all prefetch instructions injected into a basic block and groups them by context. As shown in Fig. 4.8, prefetches for addresses `0xA` and `0xD` are grouped together since they are conditional on the same context, `C0`. Similarly, `0x4`, `0x2`, and `0x7` are grouped together since they share the same context `C1`.

Next, I-SPY attempts to merge a group of prefetch instructions into a single prefetch instruction. I-SPY uses an n-bit bitmap to select a subset of cache lines within a window of n consecutive cache lines. In the example shown in Fig. 4.8, the coalesced prefetch for context `C1` has two bits set in the bitmask to encode lines `0x4` and `0x7` where the base address of the prefetch is `0x2`. While a larger bitmask allows coalescing more prefetches, it also increases hardware complexity. We study

the effect of bitmask size in Fig. 4.17.

Coalesced prefetch instruction. Our proposed coalesced prefetch instruction, `Lprefetch`, requires an additional operand for specifying the coalescing bit-vector. Prefetch instructions in current hardware (e.g., `prefetcht*` on x86 and `pli` on ARM) follow the format, (`prefetch, address`), which takes `address` as an operand and prefetches the cache line corresponding to `address`. `Lprefetch` takes an extra operand, `bit-vector`. The `prefetcht*` instruction on x86 has a size of 7 bytes, hence, with the addition of an $n = 8$ bits bitmask, `Lprefetch` has a size of 8 bytes.

I-SPY combines prefetch coalescing and conditional prefetching via another instruction, `CLprefetch`, with the format (`prefetch`, `address`, `context-hash`, `bit-vector`) as shown in Fig. 4.8. `CLprefetch` prefetches all the prefetch targets specified by `bit-vector` only if the current context matches with the context encoded in the `context-hash`. This new instruction has a size of 10 bytes (2 extra bytes to specify `context-hash`).

Micro-architectural modifications. Coalesced prefetch instructions require minor micro-architectural modifications that mainly consists of a series of simple incrementers. These incrementers decode the 8-bit coalescing vector and enable prefetching up to 9 cache lines (the initial prefetch target, plus up to 8 bit-vector-dependent targets). The resultant cache line addresses are then forwarded to the prefetch engine.

Replacement policy for prefetched lines. I-SPY's prefetch instructions also update the replacement policy priority of the prefetched cache line. Instead of assigning the highest priority to the prefetched cache line (as done for demand-loads), I-SPY's prefetch instructions assign the prefetched cache line a priority equal to the half of the highest priority. I-SPY's goal with this policy is to reduce the adverse effects of a potentially inaccurate prefetch operation.

4.4 Usage Model

We provide an overview of the high-level usage model of I-SPY in Fig. 4.9. I-SPY profiles an application's execution at run time, and uses these profiles to perform an offline analysis of I-cache misses to suitably inject code prefetch instructions.

Online profiling. I-SPY first profiles an application's execution at run time (step ①). It uses Intel's LBR [7] to construct a dynamic CFG (such as the one shown in Fig. 4.2), and augments the dynamic CFG with L1 I-cache miss profiles collected with Intel's PEBS [99] hardware performance counters. At every I-cache miss, I-SPY records the program counters of the previous 32 branches that the program executed (on x86_64, LBR has a 32-entry limit). Run-time profiling using Intel LBR's and Intel PEBS's lightweight monitoring [120, 7] enables profiling applications online, in production.

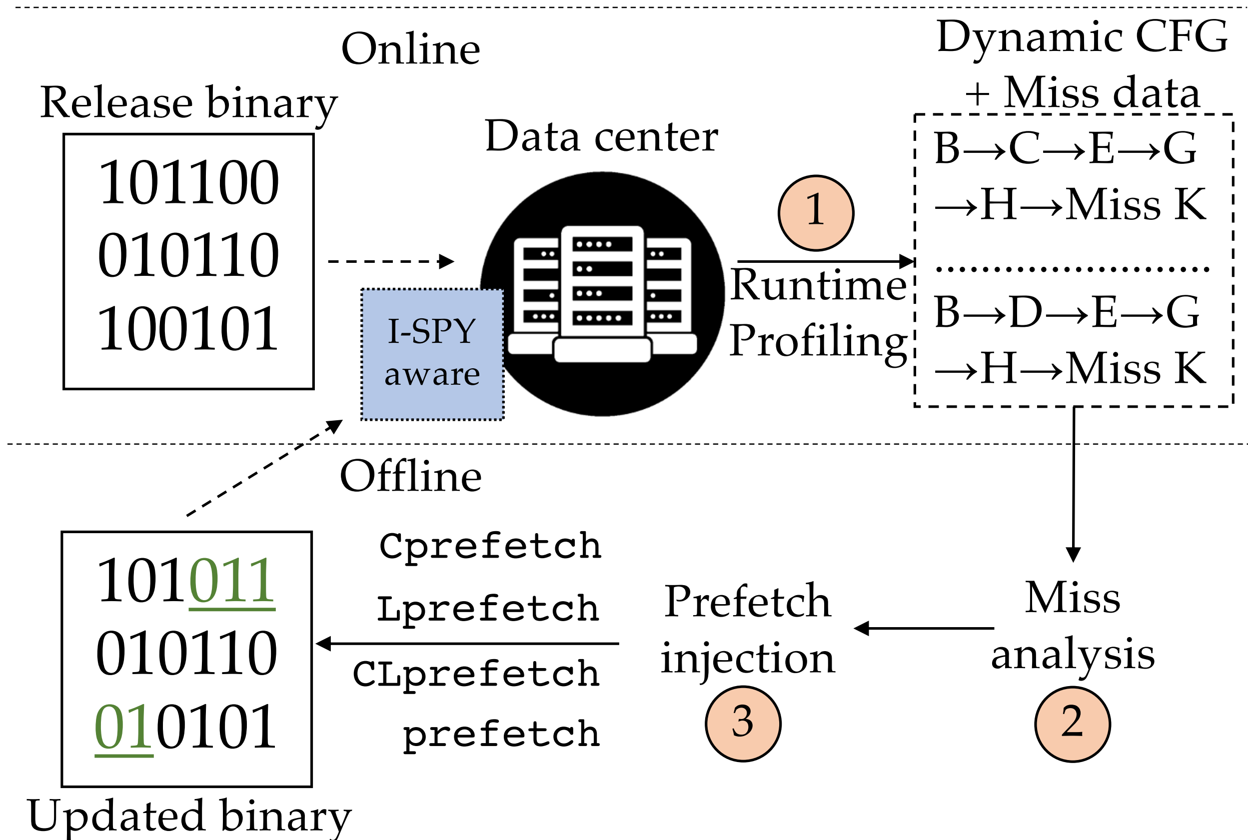


Figure 4.9: Usage model of I-SPY

Offline analysis. Next, I-SPY performs an offline analysis (②) of the miss-annotated dynamic CFG that it generates at run time. For each miss, I-SPY considers all predecessor basic blocks within the prefetch window. Unlike prior work [41], I-SPY does not require the per-application IPC metric to find predecessors within the prefetch window as the LBR profile already includes dynamic cycle information for each basic block. Apart from this, the algorithm to find the best prefetch injection site is similar to prior work [41] and has a worst-case complexity of $O(n \log n)$.

After finding the best prefetch injection site to cover each miss, I-SPY runs two extra analyses, context discovery and prefetch coalescing. First, if the prefetch injection site has a non-zero fan-out, I-SPY analyzes the predecessors of the injection site to reduce its fan-out (Fig. 4.6). Next, if the same injection site is selected for prefetching multiple cache lines, I-SPY applies prefetch coalescing to reduce the number of prefetch instructions (Fig. 4.8).

Once I-SPY finishes identifying opportunities for conditional prefetching and prefetch coalescing, it injects appropriate prefetch instructions to cover all misses. Specifically, I-SPY injects four kinds of prefetch instructions (③).

If the context of a given prefetch instruction differs from the contexts of all other prefetch instructions, then this prefetch instruction cannot be coalesced with others. In that case, I-SPY

injects a `Cprefetch` instruction.

Conditionally prefetching a line based on the execution context may not improve the prefetch accuracy. In this case, I-SPY will try to inject an `Lprefetch` instruction. If multiple cache lines are within a range of n lines (where n is the size of `bit-vector` used to perform coalescing as in §4.3.2) from the nearest prefetch target, I-SPY will inject an `Lprefetch`. Otherwise, I-SPY will inject multiple AsmDB-style `prefetch` instructions that simply prefetch a single target cache line.

If conditional prefetching improves prefetching accuracy and multiple cache lines can be coalesced, I-SPY injects `CLprefetch` instructions.

The new binary updated with code prefetch instructions is deployed on I-SPY-aware data center servers that can conditionally execute and (or) coalesce the injected prefetches.

4.5 Evaluation Methodology

We envision an end-to-end I-SPY system that uses application profile information and our proposed family of hardware code prefetch instructions. We evaluate I-SPY using simulation since existing server-class processors do not support our proposed hardware modifications for conditional prefetching and prefetch coalescing. Additionally, simulation enables replaying memory traces to conduct limit studies and compare I-SPY's performance against an ideal prefetch mechanism. We prototype the state-of-the-art prefetcher, AsmDB [41], and compare I-SPY against it. We now describe (1) the experimental setup that we use to collect an application's execution profile, (2) our simulation infrastructure, (3) I-SPY's system parameters, and (4) the data center applications we study.

Data collection. During I-SPY's offline phase, we use Intel's LBR [7] and PEBS counters [83] (more specifically (`frontend_retired.l1i_miss`)) to collect an application's execution profile and L1 I-cache miss information. We record up to 100 million instructions executed in steady-state. We combine our captured miss profiles and instruction traces to construct an application's miss-annotated dynamic CFG.

Simulation. We use the ZSim simulator [310] to evaluate I-SPY. We modify ZSim [310] to support conditional prefetching and prefetch coalescing. We use ZSim in a trace-driven execution mode, modeling an out-of-order processor. The detailed system parameters are summarized in Table 4.1. Additionally, we extend ZSim to support our family of hardware code prefetch instructions. Our implemented code prefetch instructions insert prefetched cache lines with a lower replacement policy priority than any demand load requests.

System parameters. Based on the sensitivity analysis (see Fig. 4.18), we use 27 cycles as minimum prefetch distance, and 200 cycles as maximum prefetch distance. Additionally, we empiri-

Table 4.1: Simulated System

Parameter	Value
CPU	Intel Xeon Haswell
Number of cores per socket	20
L1 instruction cache	32 KiB, 8-way
L1 data cache	32 KiB, 8-way
L2 unified cache	1 MB, 16-way
L3 unified cache	Shared 10 MiB per socket, 20-way
All-core turbo frequency	2.5 GHz
L1 I-cache latency	3 cycles
L1 D-cache latency	4 cycles
L2 cache latency	12 cycles
L3 cache latency	36 cycles
Memory latency	260 cycles
Memory bandwidth	6.25 GB/s

cally determine that coalescing non-contiguous prefetches that occur within a cache line window of 8 cache lines yields the best performance.

Data center applications. We evaluate nine popular data center applications described in Sec. 4.2. We allow an application's binary to be built with classic compiler code layout optimizations such as in-lining [38], hot/cold splitting [72], or profile-guided code alignment [278]. We study these applications with different input parameters offered to the client's load generator (e.g., number of requests per second or the number of threads).

Evaluation metrics. We use six evaluation metrics to evaluate I-SPY's effectiveness. First, we compare I-SPY's performance improvement against an ideal cache and AsmDB. Second, we study how well I-SPY reduces L1 I-cache MPKI compared to the state-of-the-art prefetcher, AsmDB [41]. Third, we analyze how much performance improvement stems from conditional prefetching and prefetch coalescing, individually. Fourth, we compare I-SPY's prefetch accuracy with AsmDB. Fifth, we analyze the static and dynamic code footprint increase induced by I-SPY. Sixth, we determine whether I-SPY achieves high performance across various application inputs. Since, data center applications often run continuously, application inputs can drastically vary (e.g., diurnal load trends or load transients [345, 324]). Hence, a profile-guided optimization for data center applications must be able to improve performance across diverse inputs.

We also perform a sensitivity analysis of I-SPY's system parameters by evaluating the effect of varying the (1) number of predecessors in `context-hash`, (2) minimum and maximum prefetch distances, (3) coalescing size, and (4) context size used to conditionally prefetch.

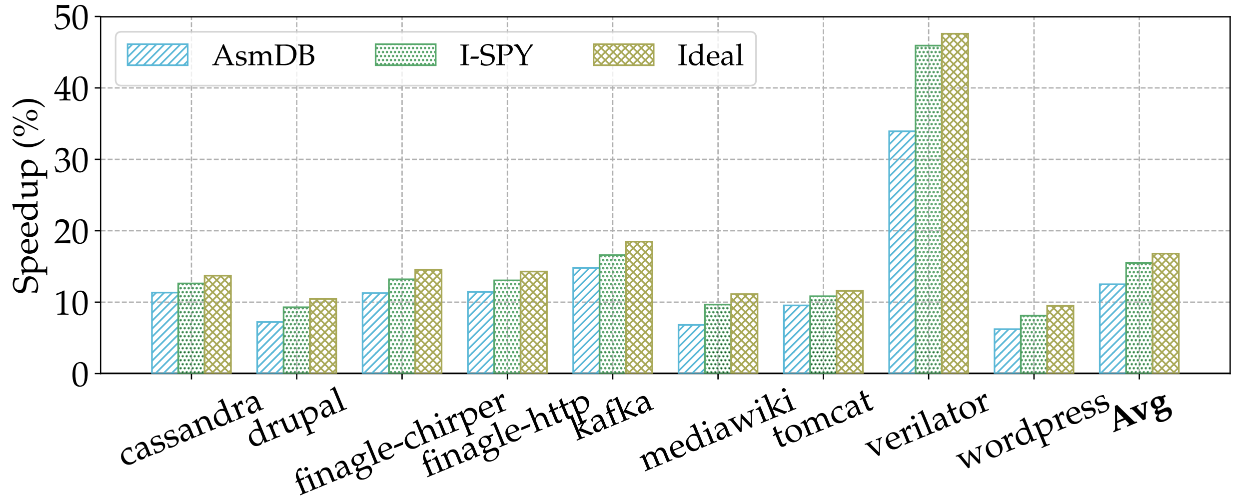


Figure 4.10: I-SPY's speedup compared to an ideal cache and AsmDB: I-SPY achieves an average speedup that is 90.4% of ideal.

4.6 Evaluation

In this section, we evaluate how I-SPY improves application performance compared to an ideal cache implementation and the state-of-the-art prefetcher [41], AsmDB, using the evaluation metrics defined in §4.5. We then perform sensitivity studies to determine the effect of varying I-SPY's configurations.

4.6.1 I-SPY: Performance Analysis

Speedup. We first evaluate the speedup achieved by I-SPY across all applications. In Fig. 4.10, we show I-SPY's speedup (green bars) compared against an ideal cache that faces no misses (brown bars) and AsmDB [41] (blue bars).

We find that I-SPY attains a near-ideal speedup, achieving an average speedup that is 90.4% (up to 96.4%) of an ideal cache that always hits in the L1 I-cache. I-SPY falls slightly short of an ideal cache since (1) it executes more instructions due to the injected prefetch instructions and (2) a previously unobserved execution context might not trigger a prefetch, precipitating a miss. Additionally, I-SPY outperforms AsmDB by 22.4% on average (up to 41.2%), since it eliminates more I-cache misses than AsmDB as we show next.

L1 I-cache MPKI reduction. We next evaluate how well I-SPY reduces L1 I-cache misses compared to AsmDB [41] in Fig. 4.11. We evaluate across all nine applications.

We observe that I-SPY achieves a high miss coverage, reducing L1 I-cache MPKI by an average of 95.8% across all applications. Furthermore, I-SPY reduces MPKI compared to AsmDB by an

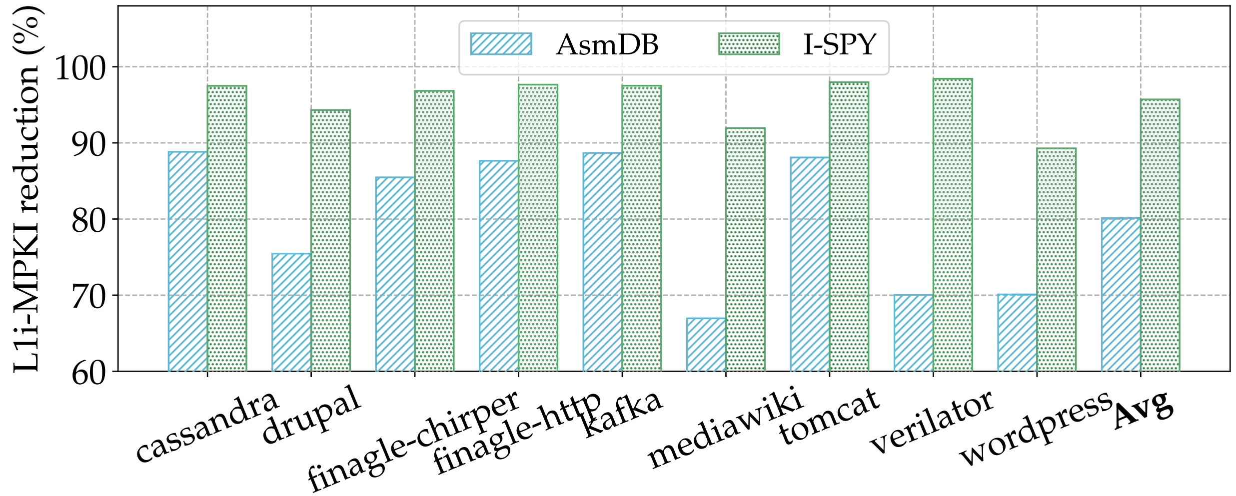


Figure 4.11: I-SPY's L1 I-cache MPKI reduction compared with AsmDB: I-SPY removes 15.7% more misses than AsmDB.

average of 15.7% across all applications (the greatest improvement is 28.4% for *verilator*). The MPKI reduction is due to conditionally executing prefetches and coalescing them, thereby eliminating more I-cache misses. In contrast, AsmDB executes a large number of unused prefetches that evict useful data from the cache.

Performance of conditional prefetching and prefetch coalescing. In Fig. 4.12, we quantify how much I-SPY's conditional prefetching and prefetch coalescing mechanisms contribute to net application speedup. We show the performance improvement achieved by these novel mechanisms over AsmDB, across all nine applications. We make two observations.

First, we note that both conditional prefetching and prefetch coalescing provide gains over AsmDB across all applications. Conditional prefetching improves performance more than coalescing for eight of our applications, since it covers more I-cache misses with better accuracy. In *verilator*, we observe that coalescing offers a better performance since 75% of *verilator*'s misses have a high spatial locality even within a cache line window of 8 lines.

Second, we find that the performance gains achieved by conditional prefetching and prefetch coalescing are not strictly additive. As I-SPY only coalesces prefetches that have the same condition, many prefetch instructions that depend on different conditions are not coalesced. Yet, combining both techniques offers better speedup than their individual counterparts.

Prefetch accuracy. We portray the prefetch accuracy achieved by I-SPY across all nine applications in Fig. 4.13. We also compare I-SPY's prefetch accuracy against AsmDB.

We find that I-SPY achieves an average of 80.3% prefetch accuracy. Furthermore, I-SPY's accuracy is 8.2% (average) better than AsmDB's accuracy, since I-SPY's conditional prefetching

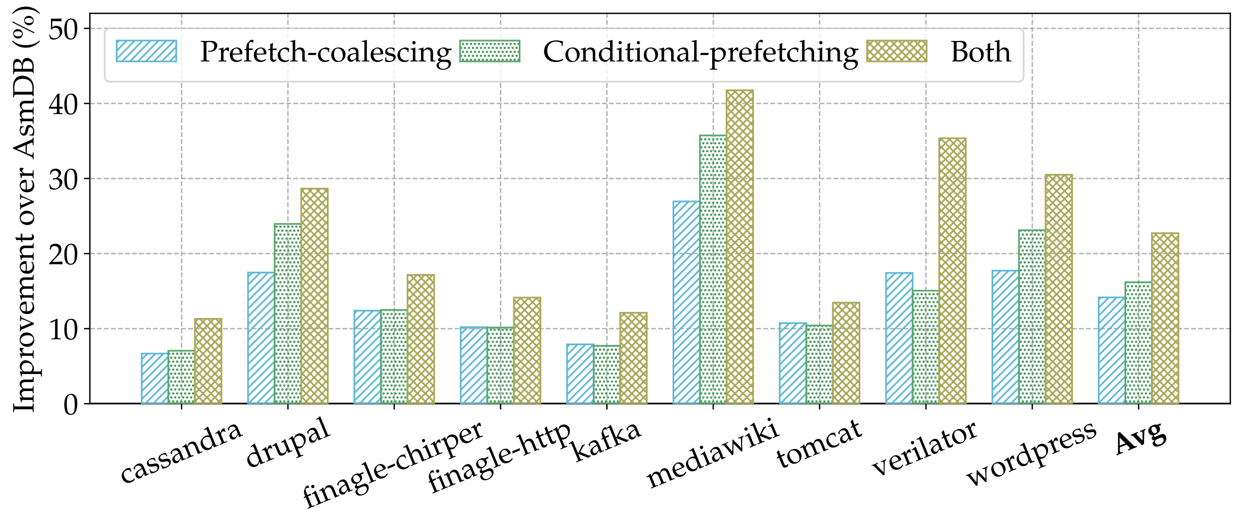


Figure 4.12: Speedup achieved by conditional prefetching and prefetch coalescing over AsmDB: Conditional prefetching often offers better speedup than coalescing, but their combined speedup is significantly better.

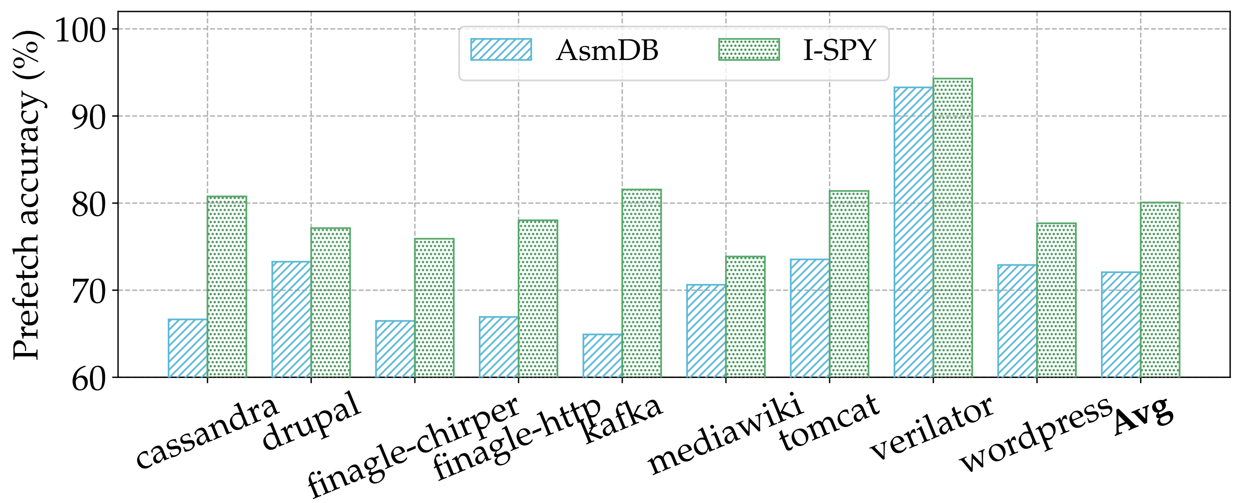


Figure 4.13: I-SPY's prefetch accuracy compared with AsmDB: I-SPY achieves an average of 8.2% better accuracy than AsmDB.

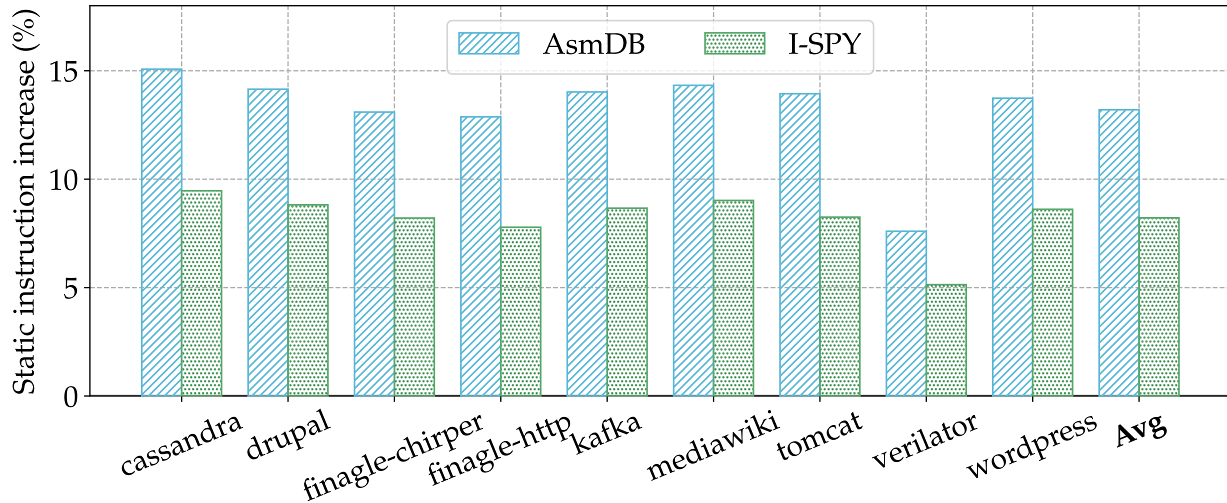


Figure 4.14: I-SPY's static code footprint increase compared to AsmDB: I-SPY statically injects 37% (average) fewer instructions than AsmDB.

avoids trading off prefetch accuracy for miss coverage, unlike AsmDB.

Static and dynamic code footprint increase. We next evaluate by how much I-SPY increases applications' static and dynamic code footprints. First, we illustrate the static code footprint increase induced by I-SPY in Fig. 4.14. We also compare against AsmDB's static code footprint.

We observe that I-SPY increases the static code footprint by 5.1% - 9.5% across all applications. By coalescing multiple prefetches into a single prefetch instruction, I-SPY introduces fewer prefetch instructions into the application's binary. In contrast, we find that AsmDB increases the static code footprint much more starkly—7.6% - 15.1%.

Next, we study by how much I-SPY increases the dynamic application footprint in Fig. 4.15 across all nine applications. We note that I-SPY executes 3.7% - 7.2% additional dynamic instructions since it covers I-cache misses by executing injected code prefetch instructions. We observe that AsmDB has a higher dynamic instruction footprint across eight applications (ranging from 5.5% - 11.6%), since it does not coalesce prefetches like I-SPY. For *verilator*, I-SPY's dynamic footprint is higher than AsmDB since I-SPY covers 28.4% more misses than AsmDB by executing more prefetch instructions, while also providing 35.9% performance improvement over AsmDB.

Generalization across application inputs. To determine whether I-SPY achieves a performance improvement with an application input that is different from the profiled input, we characterize I-SPY's performance for five different inputs fed to three of our applications—*drupal*, *mediawiki*, *wordpress* (Fig. 4.16). We choose these three applications, because they have the greatest variety of readily-available test inputs that we can run. We compare I-SPY against AsmDB in terms of ideal cache performance.

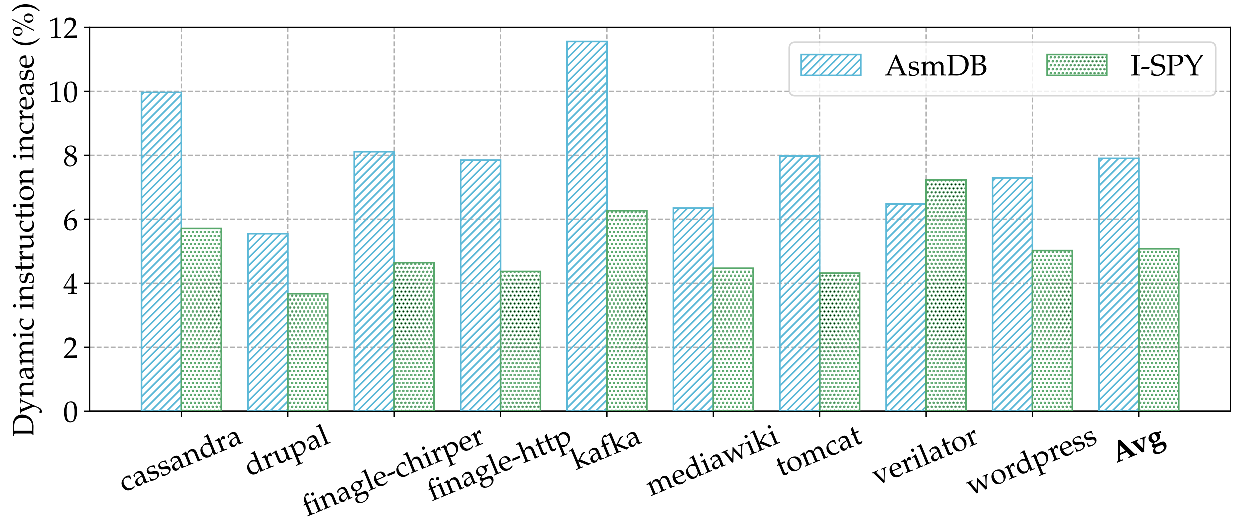


Figure 4.15: I-SPY's dynamic code footprint increase compared to AsmDB: On average, I-SPY executes 36% fewer prefetch instructions than AsmDB.

We observe that I-SPY achieves a speedup that is closer to the ideal speedup than the speedup provided by AsmDB across all test inputs. I-SPY is more resilient to the input changes than AsmDB because of conditional prefetching. I-SPY achieves at least 70% (up to 86.84%) of ideal cache performance on inputs that are different from the profiled input.

4.6.2 I-SPY: Sensitivity Analysis

We next evaluate how I-SPY's performance varies in response to variations of the different system parameters.

Number of predecessors comprising the context. In Fig. 4.17, we observe how the I-SPY conditional prefetching's performance varies in response to a variation in the number of predecessors comprising the context condition (see Sec 4.3.1). We vary predecessor counts from 1 to 32 (with a geometric progression of 2) and show the I-SPY conditional prefetching's average performance improvement across all nine applications.

We find that the I-SPY conditional prefetching's performance improves with an increase in the number of predecessors composing the context condition. Using more predecessors enables a more complete context description, and slightly improves performance by predicting I-cache misses more accurately. However, a large number of predecessors impose a significant context-discovery computation overhead. Specifically, the search space of possible predecessor candidates grows exponentially with the number of predecessors comprising the context condition. Consequently, the context discovery process takes tens of minutes to complete with more than 4 predecessors,

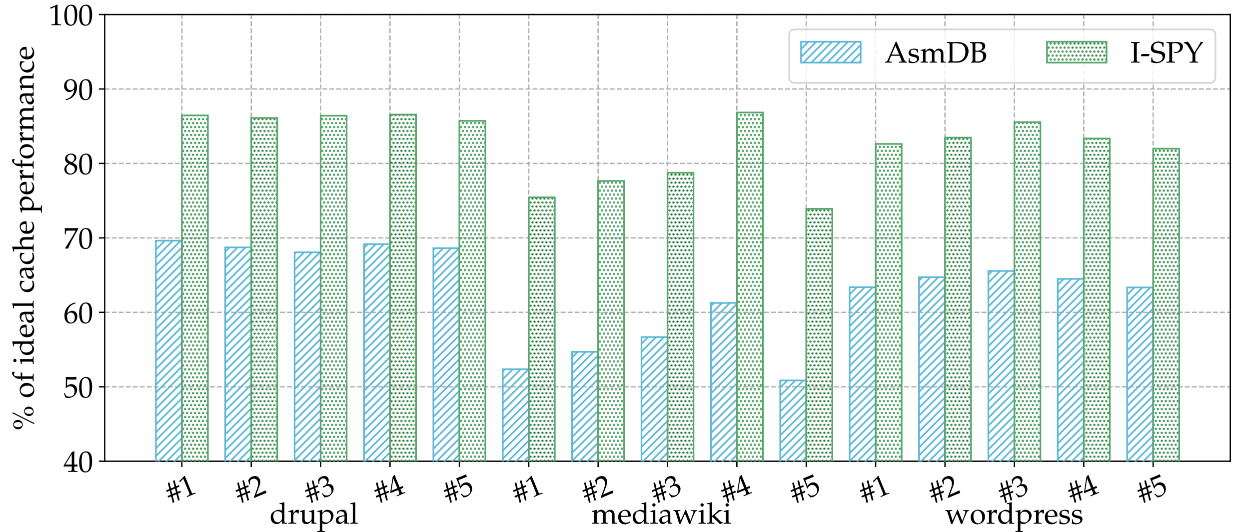


Figure 4.16: I-SPY's performance compared against AsmDB for different application test inputs: I-SPY outperforms AsmDB when the application input differs from the profiled input.

which can be a bottleneck in the build process. Since I-SPY's conditional prefetching achieves more than 85% of ideal cache performance even with four predecessors, I-SPY's design uses four predecessors to define context and keeps the computational overhead of context discovery low.

Minimum and maximum prefetch distance. We next analyze how I-SPY's performance varies with an increase in the minimum and maximum prefetch distances, in Fig. 4.18. We observe that I-SPY achieves maximum performance for a minimum prefetch distance of 20-30 cycles (which is greater than typical L2 access latency but less than L3 access latency). On the other hand, an increase in the maximum prefetch distance always improves I-SPY's performance. However, the increase is very slow after 200 cycles. Based on these results, we use 27 cycles as the minimum prefetch distance, and 200 cycles as the maximum prefetch distance for I-SPY.

Coalescing size. We next study the sensitivity of I-SPY's prefetch coalescing to the coalesce bitmask size (see §4.3.2) in Fig. 4.19. We vary the coalesce bitmask size from 1 bit to 64 bits, prefetching up to 2 and 65 cache lines using a single instruction, respectively. We then measure the percentage of ideal speedup achieved by I-SPY's prefetch coalescing as an average across all applications.

We note that I-SPY's performance improves slightly with a larger bitmask, since larger bitmasks enable coalescing more cache lines, reducing spurious evictions. However, a large bitmask will introduce hardware design complexities since the microarchitecture must now support additional in-flight prefetch operations. Similar to prior work [211], to minimize hardware complexity, we design I-SPY with an 8-bit coalescing bitmask, since it can be implemented with minor hardware modifications (as described in §4.3.2).

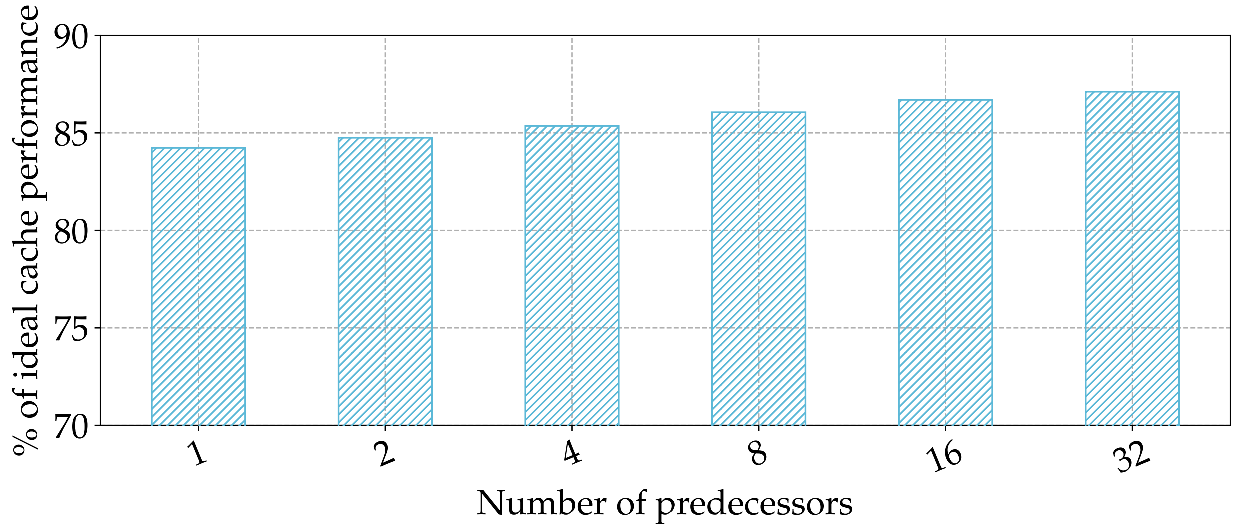


Figure 4.17: I-SPY's conditional prefetching achieves better performance with an increase in the number of predecessors comprising the context.

Additionally, we examine which and how many nearby cache lines a coalesced prefetch instruction typically prefetches for all nine applications. As shown in Fig. 4.20, the probability of coalesced prefetching reduces with an increase in cache line distance. Moreover, most coalesced prefetch instructions (82.4% averaged across nine applications) prefetch less than four cache lines.

Context hash size. We next analyze how I-SPY's false positive rate varies with an increase in the context hash size, in Fig. 4.21. We study the *wordpress* benchmark since its speedup is heavily impacted by prefetch accuracy (see Fig. 4.3).

We observe that increasing the number of bits in the context hash reduces the false positive rate. However, an increase in the context hash size increases the static code footprint, as shown in Fig. 4.21. To minimize the static code footprint while still achieving a low false positive rate, I-SPY's design uses a 16-bit context hash—13% false positive rate and 4.6% static code increase.

4.7 Discussion

In this section we discuss some limitations of I-SPY and offer potential solutions.

Prefetching already resident cache lines. Although our process of discovering high-probability contexts that lead to cache misses is effective, we also found that many times, the target cache line of a `Cprefetch` is already resident in the cache. However, the overhead of such resident prefetch operations is low since they do not poison the cache by bringing in new unnecessary cache lines. To make this overhead even lower, we design our proposed prefetch instructions such that they are always inserted with a lower priority as demand loads in regards to the replacement policy.

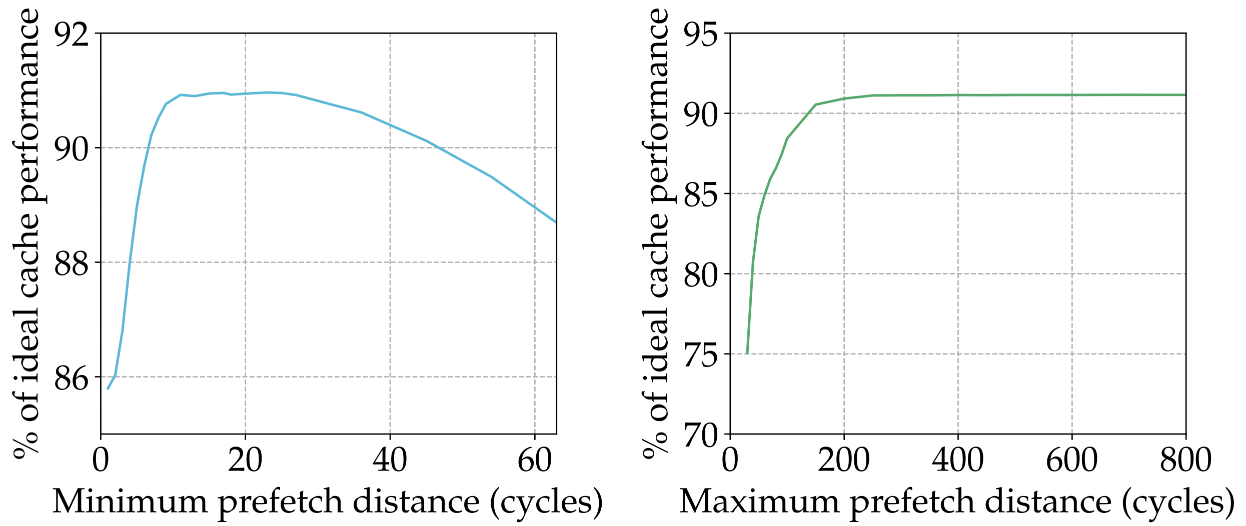


Figure 4.18: I-SPY's average performance variation in response to changes in the minimum (left) and the maximum (right) prefetch distance.

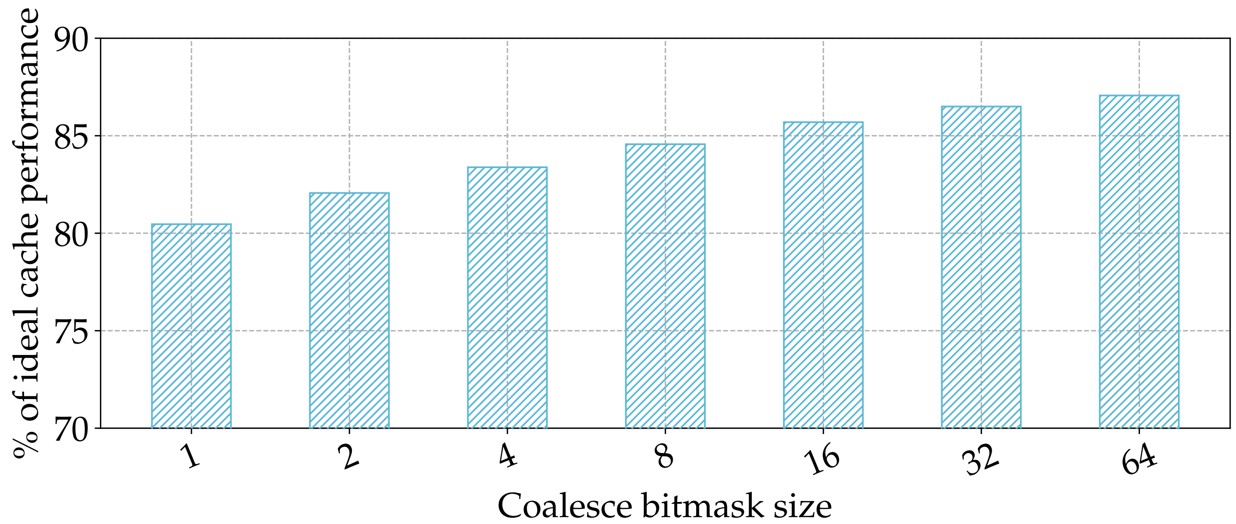


Figure 4.19: I-SPY's average performance variation in response to increasing the coalescing size: Larger coalescing sizes achieve higher gains.

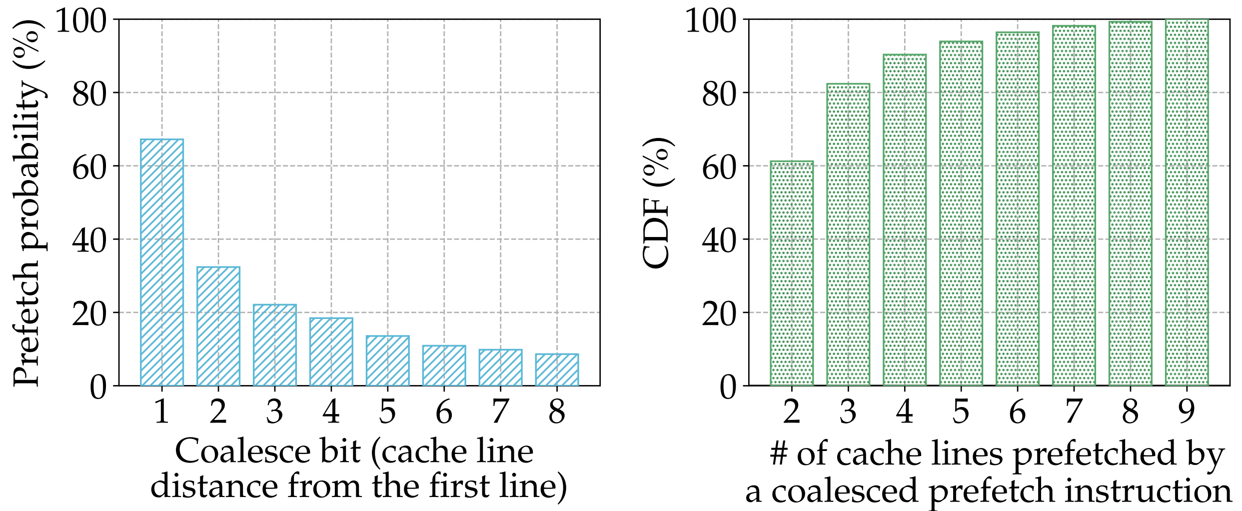


Figure 4.20: (left) The probability of coalesced prefetching reduces with an increase in cache line distance. (right) Coalesced prefetch instructions usually bring in less than 4 cache lines.

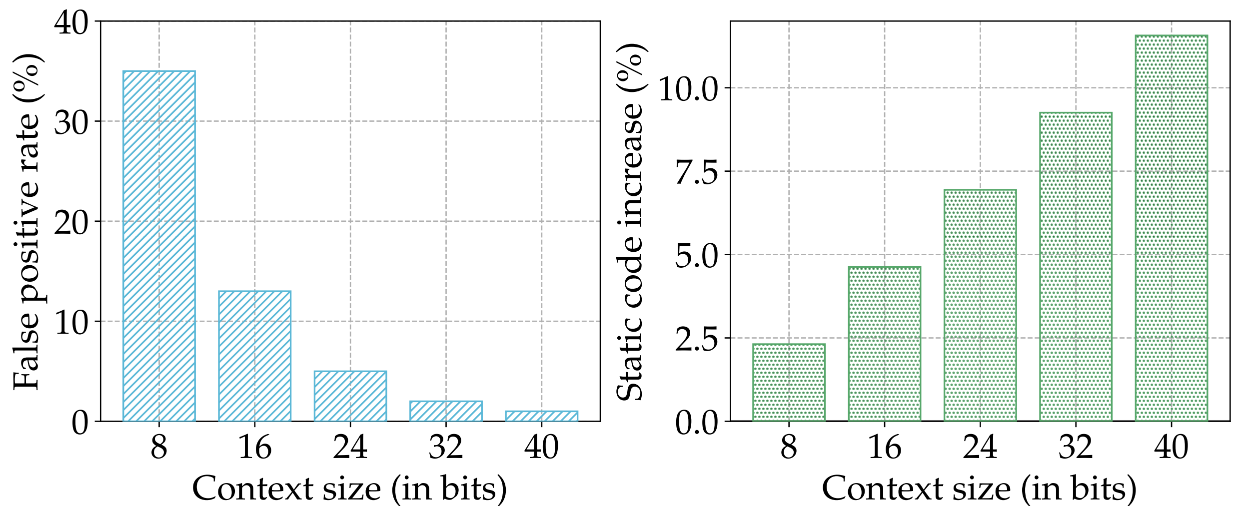


Figure 4.21: (left) I-SPY's false positive rate variation in response to an increase in context size: False positives are reduced with a larger context; (right) I-SPY's static code footprint size variation in response to context size: Static code footprint increases with an increase in context size.

Prefetching within JITted code. Most instruction cache misses in code generated at run time are out of I-SPY's scope. While I-SPY is able to prefetch for some of these misses via `Cprefetch` instructions inserted into non-JITted code, there are still up to 10% of code misses in JITted code (mostly for the three HHVM applications, *wordpress*, *drupal*, and *mediawiki*) that are not covered. To handle these additional misses, I-SPY could be integrated with a JIT compiler since all of I-SPY's offline machinery (which leverages hardware performance monitoring mechanisms) can, in principle, be used online by the runtime instead.

4.8 Related Work

The performance criticality of instruction cache misses has resulted in a rich body of prior literature. We discuss three categories of related work.

Software prefetching. Several software techniques [65, 136, 299, 410, 238, 286, 40, 137] improve instruction locality by relocating infrequently executed code via Profile-Guided Optimizations (PGO) at compile time [65], link time [221, 276], or post link time [236, 278]. However, finding the optimal cache-conscious layout is intractable in practice [41], since it requires meandering through a vast number of control-flow combinations. Hence, existing techniques must oftentimes make inaccurate control-flow approximations. Whereas PGO-based techniques have been shown to improve data center application performance [65, 278], they still eliminate only a small subset of all instruction cache misses [41].

Hardware prefetching. Hardware instruction prefetching techniques began with next-line instruction prefetchers that exploit the common case of fetching sequential instructions [31]. These next-line prefetchers soon evolved into next-N-line and instruction stream prefetchers [328, 302, 114, 280, 113, 189, 209] that use trigger events and control mechanisms to prefetch by adaptively looking a few instructions ahead. Next-line and stream prefetchers have been widely deployed in industrial designs because of their implementation simplicity. However, such next-line prefetchers are often inaccurate for complex data center applications that implement frequent branching and function calls.

Branch predictor based prefetchers [53, 60, 212, 211, 339, 343, 114] improve prefetch accuracy in branch- and call-heavy code. Run-ahead execution [265], wrong path instruction prefetching [290], and speculative prefetching mechanisms [353, 412] can also explore ahead of the instruction fetch unit. However, such prefetchers are susceptible to interference precipitated by wrong path execution and insufficient look ahead when the branch predictor traverses loop branches [113].

TIFS [113] and PIF [114] record the instruction fetch miss and instruction commit sequences to overcome the limitations of branch predictor based prefetching. Whereas these mechanisms have improved accuracy and miss coverage, they require considerable on-chip storage to maintain an

ordered log of instruction block addresses. Increasing on-chip storage is impractical at data center scale due to strict energy requirements.

More sophisticated hardware instruction prefetchers proposed by prior works (e.g., trace caches and special hardware replacement policies) [209, 306, 152, 28] are too complex to be deployed. We conclude that hardware prefetching mechanisms either provide low accuracy and coverage or they require significant on-chip storage and are too complex to implement in real hardware.

In comparison, I-SPY covers most instruction cache misses with minor micro-architectural modifications. I-SPY requires only 96-bits of extra storage while state-of-the-art hardware prefetchers (e.g., SHIFT [189], Confluence [190], and Shotgun [211]) require kilobytes to megabytes of extra storage.

Hybrid hardware-software prefetching. Hybrid hardware-software techniques [238, 41] attempt to overcome the limitations of hardware-only and software-only prefetching mechanisms. These mechanisms propose hardware code prefetch instructions [12] that are similar to existing data prefetch instructions [11]. They use software-based control flow analyses to inject hardware code prefetch instructions.

Although existing hybrid instruction prefetching mechanisms have been the most effective in reducing I-cache misses in data-center applications [41], they suffer from key limitations that hurt prefetch accuracy. First, such hybrid techniques rely on a single predecessor basic block as the execution context to predict a future cache miss. However, as we show in Section 4.2, we find that miss patterns are more complex and multiple predecessor basic blocks are needed to construct the execution context to accurately predict a future cache miss. Second, existing hybrid prefetching techniques often execute far too many dynamic prefetch instructions, further increasing application code footprints. In contrast, I-SPY achieves near-ideal prefetch accuracy via conditional prefetching, while allowing only a small increase in application footprint.

4.9 Conclusion

Large instruction working sets in modern data center applications have resulted in frequent I-cache misses that significantly degrade data center performance. We investigated instruction prefetching to address this problem and analyze the challenges of designing an ideal instruction prefetcher. We then used insights derived from our investigation to develop I-SPY, a novel profile-driven prefetching technique. I-SPY exposes two new instruction prefetching techniques: *conditional prefetching* and *prefetch coalescing* via a family of light-weight hardware code prefetch instructions. We evaluated I-SPY on nine widely-used data center applications to demonstrate an average of 15.5% (up to 45.9%) speedup and 95.9% (and up to 98.4%) reduction in instruction cache misses, outperforming the state-of-the-art prefetching technique by 22.5%.

CHAPTER 5

Ripple: Profile-Guided Instruction Cache Replacement for Data Center Applications

Modern data center applications exhibit deep software stacks, resulting in large instruction footprints that frequently cause instruction cache misses degrading performance, cost, and energy efficiency. Although numerous mechanisms have been proposed to mitigate instruction cache misses, they still fall short of ideal cache behavior, and furthermore, introduce significant hardware overheads. We[1] first investigate why existing I-cache miss mitigation mechanisms achieve sub-optimal performance for data center applications. We find that widely-studied instruction prefetchers fall short due to wasteful prefetch-induced cache line evictions that are not handled by existing replacement policies. Existing replacement policies are unable to mitigate wasteful evictions since they lack complete knowledge of a data center application's complex program behavior.

To make existing replacement policies aware of these eviction-inducing program behaviors, we propose Ripple, a novel software-only technique that profiles programs and uses program context to inform the underlying replacement policy about efficient replacement decisions. Ripple carefully identifies program contexts that lead to I-cache misses and sparingly injects "cache line eviction" instructions in suitable program locations at link time. We evaluate Ripple using nine popular data center applications and demonstrate that Ripple enables any replacement policy to achieve speedup that is closer to that of an ideal I-cache. Specifically, Ripple achieves an average performance improvement of 1.6% (up to 2.13%) over prior work due to a mean 19% (up to 28.6%) I-cache miss reduction.

[1]Some of the work in this chapter was performed in collaboration with Dexin Zhang, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci [201]. Therefore, I use the "we" pronoun in this chapter to acknowledge their involvement in this work.

5.1 Introduction

Modern data center applications are becoming increasingly complex. These applications are composed of deep and complex software stacks that include various kernel and networking modules, compression elements, serialization code, and remote procedure call libraries. Such complex code stacks often have intricate inter-dependencies, causing millions of unique instructions to be executed to serve a single user request. As a result, modern data center applications face instruction working set sizes that are several orders of magnitude larger than the instruction cache (I-cache) sizes supported by today's processors [179, 41].

Large instruction working sets precipitate frequent I-cache misses that cannot be effectively hidden by modern out-of-order mechanisms, manifesting as glaring stalls in the critical path of execution [211]. Such stalls deteriorate application performance at scale, costing millions of dollars and consuming significant energy [344, 41]. Hence, eliminating instruction misses to achieve even single-digit percent speedups can yield immense performance-per-watt benefits [344].

I-cache miss reduction mechanisms have been extensively studied in the past. Several prior works proposed next-line [328, 34, 320], branch-predictor-guided [302, 212, 211], or history-based [114, 280, 113, 189, 209, 254, 305, 128] hardware instruction prefetchers and others designed software mechanisms to perform code layout optimizations for improving instruction locality [236, 221, 65, 276, 278, 279]. Although these techniques are promising, they (1) require additional hardware support to be implemented on existing processors and (2) fall short of the ideal I-cache behavior, *i.e.*, an I-cache that incurs no misses. To completely eliminate I-cache misses, it is critical to first understand: why do existing I-cache miss mitigation mechanisms achieve sub-optimal performance for data center applications? How can we further close the performance gap to achieve near-ideal application speedup?

To this end, we comprehensively investigate why existing I-cache miss mitigation techniques fall short of an ideal I-cache, and precipitate significant I-cache Misses Per Kilo Instruction (MPKI) in data center applications (§5.2). Our investigation finds that the most widely-studied I-cache miss mitigation technique, instruction prefetching, still falls short of ideal I-cache behavior. In particular, existing prefetchers perform many unnecessary prefetches, polluting the I-cache, causing wasteful evictions. Since wasteful evictions can be avoided by effective cache replacement policies, we study previous proposals such as the Global History Reuse Predictor (GHRP) [28] (the only replacement policy specifically targeting the I-cache, to the best of our knowledge) as well as additional techniques that were originally proposed for data caches, such as Hawkeye [153]/Harmony [154], SRRIP [156], and DRRIP [156].

Driven by our investigation results, we propose Ripple, a profile-guided technique to optimize I-cache replacement policy decisions for data center applications. Ripple first performs an offline

analysis of the basic blocks (*i.e.*, sequence of instructions without a branch) executed by a data center application, recorded via efficient hardware tracing (*e.g.*, Intel's Processor Trace [207, 86]). For each basic block, Ripple then determines the cache line that an ideal replacement policy would evict based on the recorded basic block trace. Ripple computes basic blocks whose executions likely signal a future eviction for an ideal replacement policy. If this likelihood is above a certain threshold (which we explore and determine empirically in §5.3), Ripple injects an invalidation instruction to evict the victim cache line. Intel recently introduced such an invalidation instruction — `CLDemote`, and hence Ripple can readily be implemented on upcoming processors.

We evaluate Ripple in combination with I-cache prefetching mechanisms, and show that Ripple yields on average 1.6% (up to 2.13%) improvement over prior work as it reduces I-cache misses by on average 19% (up to 28.6%). As Ripple is primarily a software-based technique, it can be implemented on top of any replacement policy that already exists in hardware. In particular, we evaluate two variants of Ripple. Ripple-Least Recently Used (LRU) is optimized for highest performance and reduces I-cache MPKI by up to 28.6% over previous proposals, including Hawkeye/Harmony, DRRIP, SRRIP, and GHRP. On the other hand, Ripple-Random is optimized for lowest storage overhead, eliminating all meta data storage overheads, while outperforming prior work by up to 19%. Ripple executes only 2.2% extra dynamic instructions and inserts only 3.4% new static instructions on average. In summary, we show that Ripple provides significant performance gains compared to the state-of-the-art I-cache miss mitigation mechanisms while minimizing the meta data storage overheads of the replacement policy.

In summary, we make the following contributions:

- A detailed analysis of why existing I-cache miss mitigation mechanisms fall short for data center applications
- Profile-guided replacement: A software mechanism that uses program behavior to inform replacement decisions
- Ripple: A novel profile-guided instruction cache miss mitigation mechanism that can readily work on any existing replacement policy
- An evaluation demonstrating Ripple's efficacy at achieving near-ideal application speedup.

5.2 Why do existing I-cache miss mitigation techniques fall short?

In this section, we analyze why existing techniques to mitigate I-cache misses fall short, precipitating high miss rates in data center applications. We first present background information on the data center applications we study (§5.2.1). We then perform a limit study to determine the maximum

speedup that can be obtained with an ideal I-cache for applications with large instruction footprints (§5.2.2). Next, we evaluate existing prefetching mechanisms, including next-line prefetcher and FDIP [302], to analyze why these techniques achieve sub-optimal performance (§5.2.3). Finally, we analyze existing cache replacement policies, including LRU, Harmony, DRRIP, SRRIP, and GHRP, to quantify their performance gap with the optimal replacement policy (§5.2.4). This analysis provides the foundation for Ripple, a novel prefetch-aware I-cache replacement policy that achieves high performance with minimal hardware overheads.

5.2.1 Background on evaluated applications

We study nine widely-used real-world data center applications that suffer from substantial I-cache misses [199]—these applications lose 23-80% of their pipeline slots due to frequent I-cache misses. We study three HHVM applications from Facebook's OSS-performance benchmark suite [19], including `drupal` [378] (a PHP content management system), `mediawiki` [380] (a wiki engine), and `wordpress` [383] (a popular content management system). We investigate three Java applications from the DaCapo benchmark suite [51], including `cassandra` [2] (a NoSQL database used by companies like Netflix), `kafka` [377] (a stream processing system used by companies like Uber), and `tomcat` [4] (Apache's implementation of Java Servlet and Websocket). From the Java Renaissance [294] benchmark suite, we analyze *Finagle-Chirper* (Twitter's microblogging service) and *Finagle-HTTP* [15] (Twitter's HTTP server). We also study *Verilator* [16, 35] (used by cloud companies for hardware simulation). We describe our complete experimental setup and simulation parameters in §5.4.

5.2.2 Ideal I-cache: The theoretical upper bound

An out-of-order processor's performance greatly depends on how effectively it can supply itself with instructions. Therefore, these processors use fast dedicated I-caches that can typically be accessed in 3-4 cycles [329]. To maintain a low access latency, modern processors typically have small I-cache sizes (***e.g.***, 32KB) that are overwhelmed by data center applications' multi-megabyte instruction footprints [179, 39, 41, 279] incurring frequent I-cache misses. To evaluate the true cost of these I-cache misses as well as the potential gain of I-cache optimizations, we explore the speedup that can be obtained for data center applications with an ideal I-cache that incurs no misses. Similar to prior work [41, 199], we compute the speedup relative to a baseline cache configuration with no prefetching and with an LRU replacement policy. As shown in Fig. 5.1, an ideal I-cache can provide between 11-47% (average of 17.7%) speedup over the baseline cache configuration.

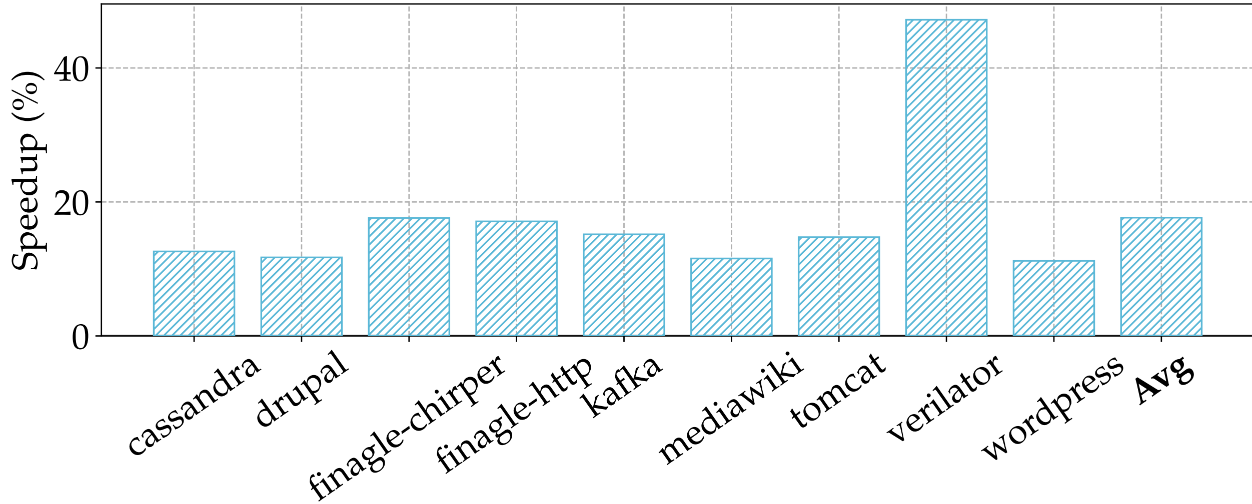


Figure 5.1: Ideal I-cache speedup over an LRU baseline without any prefetching: These data center applications can gain on average 17.7% speedup with an ideal I-cache with no misses.

5.2.3 Why do modern instruction prefetchers fall short?

Prior works [109, 41, 199, 302] have proposed prefetching techniques to overcome the performance challenge induced by insufficiently sized I-caches. Fetch Directed Instruction Prefetching (FDIP) [302] is the state-of-the art mechanism that is implemented on multiple real-world processors [350, 283, 308, 129] due to its performance and moderate implementation complexity. Fig. 5.2 shows FDIP's speedup over the baseline I-cache configuration without any prefetching. Both FDIP and baseline configurations use the LRU replacement policy. As shown, FDIP+LRU provides between 8-44% (average of 13.4%) speedup over the baseline. This represents a 4.3% performance loss over the ideal cache speedup (17.7%).

To analyze why FDIP falls short of delivering ideal performance, we equip the I-cache with a prefetch-aware ideal replacement policy. In particular, when leveraging a revised version of the Demand-MIN prefetch-aware replacement policy [154], we find that the speedup increases to on average 16.6% falling short of the ideal cache by just 1.14%. In other words, FDIP with prefetch-aware ideal replacement policy outperforms FDIP with LRU by 3.16%. This observation highlights the importance of combining state-of-the-art I-cache prefetching mechanisms with better replacement policies.

To confirm the generality of our observation, we repeat the above experiment with a standard Next-Line Prefetcher (NLP) [328]. We find that the combination of NLP prefetching with ideal cache replacement results in a 3.87% speedup over the NLP baseline without a perfect replacement policy.

To understand the key reasons behind the near-ideal speedups provided by the prefetch-aware

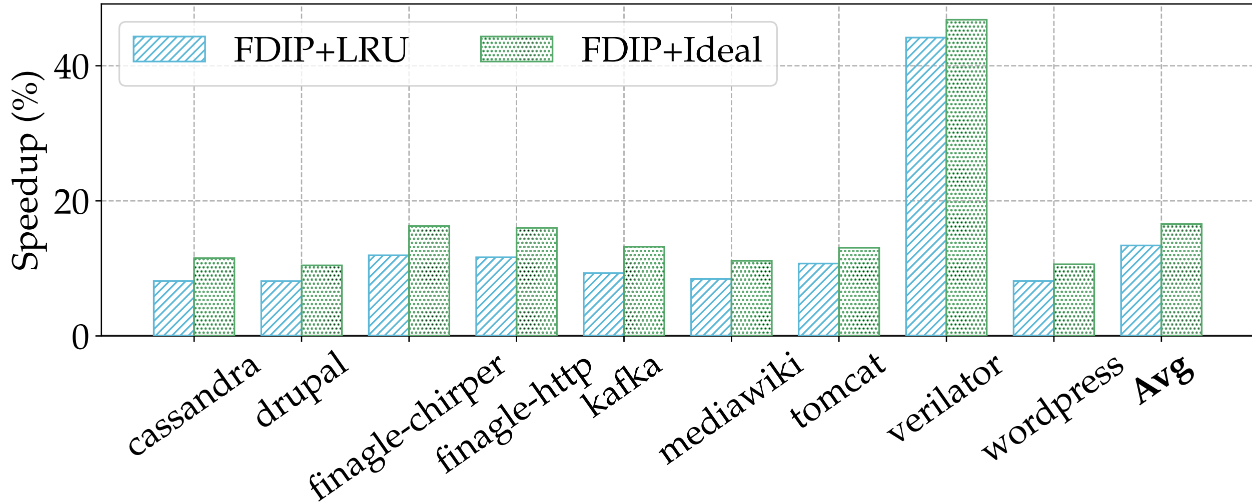


Figure 5.2: Fetch directed instruction prefetching (FDIP) speedup over an LRU baseline without any prefetching: FDIP provides 13.4% mean speedup with LRU replacement policy. However, with an ideal cache replacement policy FDIP can provide 16.6% average speedup which is much closer to ideal cache speedup.

ideal replacement policy, we first briefly describe how the policy works and then summarize the key reasons for near-ideal speedups [154]. We also quantify the speedups that an ideal replacement policy can provide for the data center applications we evaluate.

Prefetch-aware ideal replacement policy. Our ideal prefetch-aware replacement policy is based on a revised version of Demand-MIN [154]. In its revised form, Demand-MIN evicts the cache line that is prefetched farthest in the future if there is no earlier demand access to that line. If there exists no such prefetch for a given cache set, Demand-MIN evicts the line whose demand access is farthest in the future. We now detail and quantify two observations that were originally made by Demand-MIN: (1) evicting inaccurately prefetched cache lines reduces I-cache misses and (2) not evicting hard-to-prefetch cache lines reduces I-cache misses.

Observation #1: Early eviction of inaccurately prefetched cache lines reduces I-cache misses. The ideal replacement policy can evict inaccurately prefetched cache lines (*i.e.*, ones that will not be used) early, improving performance. Like most practical prefetchers, FDIP inaccurately prefetches many cache lines as its decisions are guided by a branch predictor, which occasionally mispredicts branch outcomes. However, the ideal replacement policy has knowledge of all future accesses, so it can immediately evict inaccurately prefetched cache lines, minimizing their negative performance impact. Across our nine data center applications, the ideal cache replacement policy combined with FDIP, provides 1.35% average speedup (out of 3.16% total speedup of FDIP+ideal over FDIP+LRU) relative to an LRU-based baseline replacement policy (also combined with FDIP) due to the early eviction of inaccurately-prefetched cache lines.

Observation #2: Not evicting hard-to-prefetch cache lines reduces I-cache misses. An ideal replacement policy can keep hard-to-prefetch cache lines in the cache while evicting easy-to-prefetch lines. Cache lines that cannot be prefetched with good accuracy or at all, are considered hard-to-prefetch cache lines. For example, FDIP is guided by the branch predictor. A cache line that will be prefetched based on the outcome of a branch, may not be prefetched if the predictor cannot easily predict the branch outcome (*e.g.*, due to an indirect branch)—in that case, the line is hard-to-prefetch. Easy-to-prefetch cache lines are cache lines that the prefetcher is often able to prefetch accurately. For example, a cache line that FDIP can prefetch based on the outcome of a direct unconditional branch is an easy-to-prefetch cache line. Since the ideal replacement policy has knowledge of all accesses and prefetches, it can (1) accurately identify hard-to-prefetch and easy-to-prefetch lines for any given prefetching policy and (2) prioritize the eviction of easy-to-prefetch lines over hard-to-prefetch lines. Across our nine data center applications, the ideal cache replacement policy combined with FDIP, provides 1.81% average speedup (out of 3.16% total speedup of FDIP+ideal over FDIP+LRU) relative to an LRU-based baseline replacement policy (also combined with FDIP) due to not evicting hard-to-prefetch lines.
Summary: Exploiting the above observations for an optimized prefetch-aware replacement policy requires knowledge about future instruction sequences that are likely to be executed. We find that this information can be provided by the static control-flow analysis based on execution profiles and instruction traces. As described in §5.4, Ripple leverages these analysis techniques and performs well-informed replacement decisions in concert with the prefetcher, to achieve near-ideal performance.

5.2.4 Why do existing replacement policies fall short?

In the previous section, we demonstrated that a prefetch-aware ideal cache replacement policy can provide on average 3.16% speedup relative to a baseline LRU replacement policy. In this section, we explore the extent to which existing replacement policies close this speedup gap. As there exist only few works on I-cache replacement policies apart from GHRP [28], we also explore data cache replacement policies such as LRU [249], Hawkeye [153]/Harmony [154], SRRIP [156], and DRRIP [156] applied to the I-cache.

GHRP [28] was designed to eliminate I-cache and Branch Target Buffer (BTB) misses. During execution, GHRP populates a prediction table indexed by control flow information to predict whether a given cache line is *dead* or *alive*. While making replacement decisions, GHRP favors evicting lines that are more likely to be dead. Every time GHRP evicts a cache line, it uses a counter to update the predictor table that the evicted cache line is more likely to be dead. Similarly, GHRP updates the predictor table after each hit in the I-cache to indicate that that the hit cache line is more

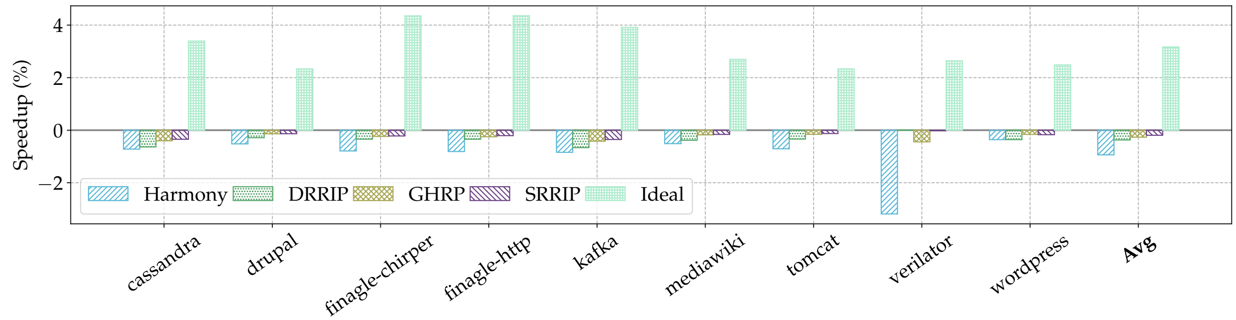


Figure 5.3: Speedup for different cache replacement policies over an LRU baseline with FDIP at the L1 I-cache: None of the existing policies outperform LRU, although an ideal replacement policy provides on average 3.16% speedup.

likely to be alive. GHRP uses 4.13KB extra on-chip metadata for a 32KB I-cache to primarily store this prediction table.

Hawkeye/Harmony [153] was designed for the data cache, specifically for the Last Level Cache (LLC). By simulating the ideal cache replacement policy [48] on access history, Hawkeye determines whether a Program Counter (PC) is "cache-friendly" or "cache-averse", *i.e.*, whether the data accessed while the processor executes the instruction corresponding to this PC follows a cache-friendly access pattern [155] or not. Cache lines accessed at a cache-friendly PC are maintained using the LRU cache replacement policy, while lines accessed by a cache-averse PC are marked to be removed at the earliest opportunity. Harmony [154] is a state-of-the-art replacement policy that adds prefetch-awareness to Hawkeye. It simulates Demand-MIN [154] on the access history in hardware to further categorize PCs as either prefetch-friendly or prefetch-averse.

SRRIP [156] was mainly designed to eliminate the adverse effects of the *scanning* [47] cache access pattern, where a large number of cache lines are accessed without any temporal locality (*i.e.*, a sequence of accesses that never repeat). SRRIP assumes that all newly-accessed cache lines are cache-averse (*i.e.*, scans). Only when a cache line is accessed for a second time, SRRIP promotes the status of the line to cache-friendly.

DRRIP [156] improves over SRRIP by considering thrashing access patterns, *i.e.*, when the working set of the application exceeds the cache size [94]. DRRIP reserves positions for both cache-friendly and cache-averse lines via set-dueling [297].

Fig. 5.3 shows the performance for different cache replacement policies over the LRU baseline with FDIP. Tab. 5.1 shows the metadata storage overheads induced by each replacement policy. As shown, none of the existing replacement policies provide any performance or storage benefits over LRU even though the ideal cache replacement policy provides 3.16% average speedup over LRU. We now explain why each of these prior replacement policies do not provide any significant benefit.

Table 5.1: Storage overheads of different replacement policies for a 32KB, 8-way set associative instruction cache that has 64B cache lines.

Replacement Policy	Overhead	Notes
LRU	64B	1-bit per line
GHRP	4.13KB	3KB prediction table, 64B prediction bits , 1KB signature, 2B history register
SRRIP	128B	2-bits$\times$ associativity
DRRIP	128B	2-bits$\times$ associativity
Hawkeye/Harmony	5.1875KB	1KB sampler (200 entries), 1KB occupancy vector, 3KB predictor, 192B RRIP counters

GHRP classifies cache lines into dead or alive based on the prediction table, to inform eviction decisions. One issue with GHRP is that it increases the classification confidence in the prediction table after eviction even if the decision was incorrect (*e.g.*, evicted a line that was still needed). We modified GHRP so that it decreases the confidence in the prediction table after each eviction. With this optimization, GHRP outperforms LRU by 0.1%.

Hawkeye/Harmony predicts whether a PC is likely to access a cache-friendly or cache-averse cache line. This insight works well for D-caches where an instruction at a given PC is responsible for accessing many D-cache lines that exhibit similar cache-friendly or cache-averse access patterns. However, for I-cache, an instruction at a given PC is responsible for accessing just one cache line that contains the instruction itself. If the line has multiple cache-friendly accesses followed by a single cache-averse access, Hawkeye predicts the line as cache-friendly. Therefore, Hawkeye cannot identify that single cache-averse access and cannot adapt to dynamic I-cache behavior. For I-cache accesses in data center applications, Hawkeye predicts almost all PCs (more than 99%) as cache friendly and hence fails to provide performance benefits over LRU.

SRRIP and DRRIP can provide significant performance benefits over LRU if the cache accesses follow a scanning access pattern. Moreover, DRRIP provides further support for thrashing access patterns [94]. For the I-cache, scanning access patterns are rare and hence classifying a line as a scan introduces a penalty over plain LRU. We quantify the scanning access pattern for our data center applications by measuring the compulsory MPKI (misses that happen when a cache line is accessed for the first time [142]). For these applications, compulsory MPKI is very small (0.1-0.3 and 0.16 on average). Moreover, both SRRIP and DRRIP arbitrarily assume that all cache lines will have similar access patterns (either scan or thrash) which further hurts data center applications' I-cache performance. Consequently, SRRIP and DRRIP cannot outperform LRU for I-cache accesses in data center applications.

We observe that data center applications tend to exhibit a unique reuse distance behavior, *i.e.*,

the number of unique cache lines accessed in the current associative set between two consecutive accesses to the same cache line, or the re-reference interval [156] of a given cache line varies widely across the program life time. Due to this variance, a single I-cache line can be both cache-friendly and cache-averse at different stages of the program execution. Existing works do not adapt to this dynamic variance and hence fail to improve performance over LRU. We combine these insights with our observations in §5.2.3 to design Ripple, a profile-guided replacement policy for data center applications.

5.3 The Ripple Replacement Mechanism

As we show in our analysis, an ideal cache replacement policy provides on average 3.16% speedup over an LRU I-cache for data center applications. Moreover, we find that existing instruction and data cache replacement policies[28, 154, 153, 156] fall short of the LRU baseline, since they are ineffective at avoiding wasteful evictions due to the complex instruction access behaviors. Hence, there is a critical need to assist the underlying replacement policy in making smarter eviction decisions by informing it about complex instruction accesses.

To this end, we propose augmenting existing replacement mechanisms with Ripple—a novel profile-guided replacement technique that carefully identifies program contexts leading to I-cache misses and strives to evict the cache lines that would be evicted by the ideal policy. Ripple's operation is agnostic of the underlying I-cache replacement policy. It sparingly injects "cache line eviction" instructions in suitable program locations at link time to assist an arbitrary replacement policy implemented in hardware. Ripple introduces no additional hardware overhead and can be readily implemented on soon-to-be-released processors. Ripple enables an existing replacement policy to further close the performance gap in achieving the ideal I-cache performance.

Fig. 5.4 shows Ripple's design components. First, at run time (online), Ripple profiles a program's basic block execution sequence using efficient hardware-based control flow tracing support such as Intel PT [207] or Last Branch Record (LBR) [99] (step ①, §5.3.1). Ripple then analyzes the program trace offline using the ideal I-cache replacement policy (step ②, §5.3.2) to compute a set of *cue blocks*. A *cue block* is a basic block whose execution almost always leads to the ideal *victim* cache line to be evicted. The key idea behind Ripple's analysis is to mimic an ideal policy that would evict a line that will be used farthest in the future. During recompilation, Ripple then injects an instruction in the cue block that invalidates the victim line (step ③ §5.3.3). Consequently, the next time a cache line needs to be inserted into the cache set that the victim line belongs to, the victim line will be evicted. In contrast to prior work [28, 153, 154, 156], Ripple moves the compute-intensive task of identifying the victim line from the hardware to the software, thereby reducing hardware overheads. We now describe Ripple's three components.

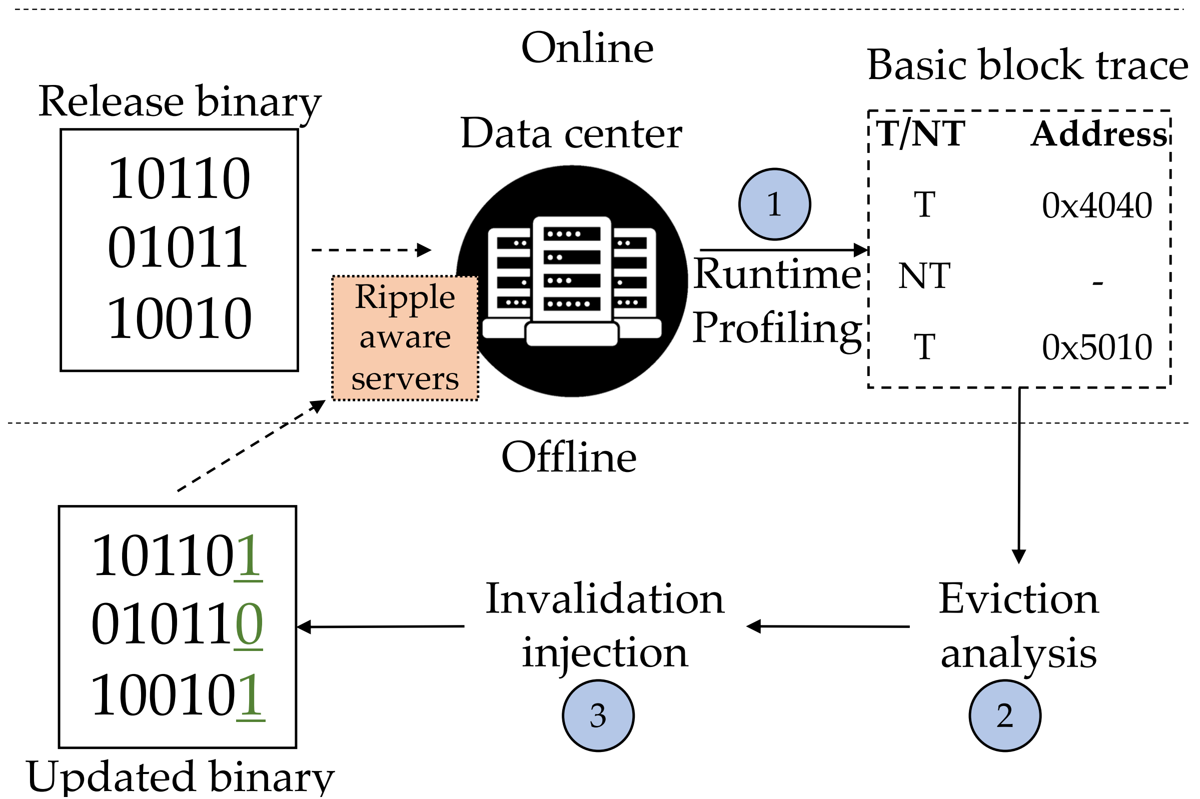


Figure 5.4: High-level design of Ripple

5.3.1 Runtime Profiling

Ripple profiles data center applications at run time using Intel PT [207] to collect a trace of the dynamically-executed basic blocks. As shown in Fig. 5.4, the collected program trace includes two pieces of information for each control-flow instruction in the program. The first bit (`T/NT`), denotes whether the branch in question was taken (T) or the fall-through path was followed (NT). If the program follows the taken path of an indirect branch, the program trace also includes the address of the next instruction on the taken path. Ripple leverages this program execution trace, to perform (1) eviction analysis and (2) invalidation injection offline at link-time. During eviction analysis, Ripple identifies the I-cache lines that will be touched (omitting speculative accesses) in real hardware based on the execution trace. Ripple's eviction analysis does not require recording the I-cache lines that will be evicted in hardware.

Ripple leverages Intel PT [207] to collect the precise basic block execution order with low runtime performance overhead (less than 1% [184, 413]). Ripple uses Intel PT since it is efficient in real-world production scenarios [86, 121, 187, 186].

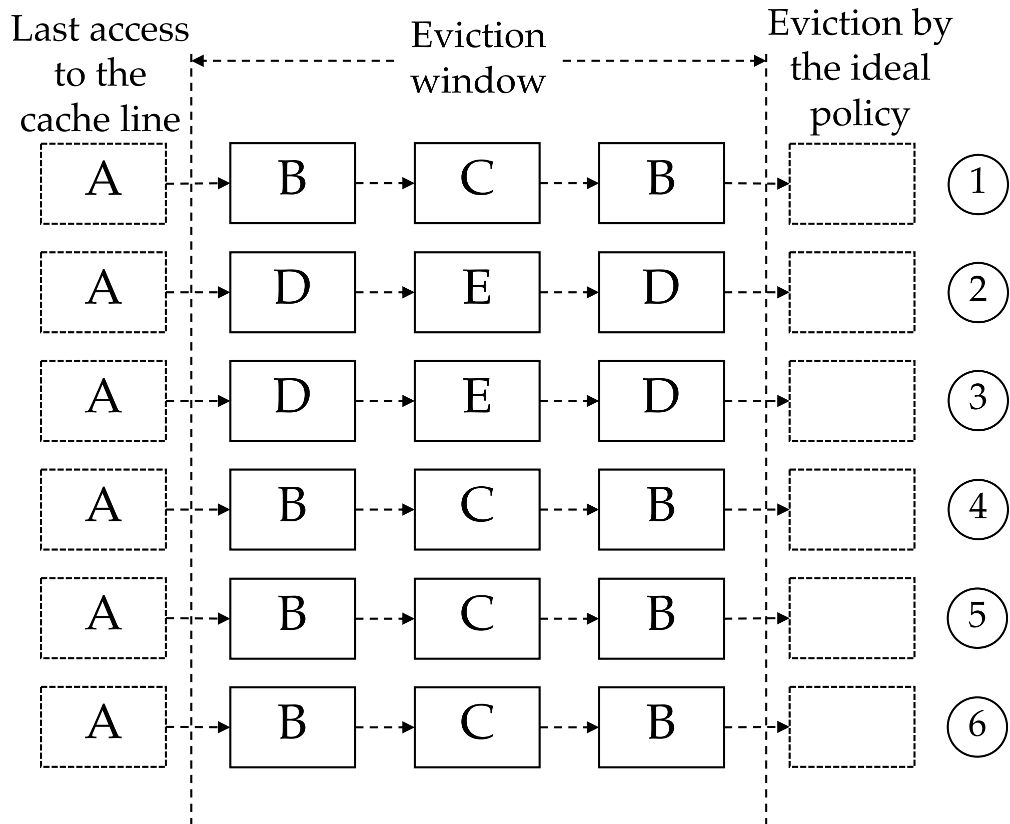
5.3.2 Eviction Analysis

The goal of Ripple's eviction analysis is to mimic an ideal replacement policy, which would evict cache lines that will not be accessed for the longest time in the future. The basic block trace collected at run time allows Ripple to retroactively determine points in the execution that would benefit from invalidating certain cache lines to help with cache replacement.

Eviction analysis determines a *cue block*, whose execution can identify the eviction of a particular *victim* cache line with high probability, if an ideal cache replacement policy were used. To determine the cue block in the collected runtime profile, Ripple analyzes all the blocks in the *eviction window* of each cache line, *i.e.*, the time window spanning between the last access to that cache line to the access that would trigger the eviction of the same line, given an ideal replacement policy.

Fig. 5.5a shows examples of eviction windows for the cache line, A. In this example, the cache line A gets evicted six times by the ideal cache replacement policy over the execution of the program. To compute each eviction window, Ripple iterates backward in the basic block trace from each point where A would be evicted by an ideal replacement policy until it reaches a basic block containing (even partially) the cache line A. Ripple identifies the basic blocks across all eviction windows that can accurately signal the eviction as *candidate cue blocks* (described further in Sec. 5.3.3), where it can insert an invalidation instruction to mimic the ideal cache replacement behavior. In this example, Ripple identifies basic blocks, B, C, D, and E as candidate cue blocks.

Next, Ripple calculates the conditional probability of a cache line eviction given the execution



(a) Cache line A's eviction window includes all basic blocks executed since A's last execution until A's eviction by the ideal cache replacement policy.

Basic block	Total executed	# of eviction windows where basic block is executed at least once	P(Eviction | Basic block)
B	16	4	0.25
C	8	4	0.5
D	6	2	0.33
E	3	2	0.66

(b) How Ripple calculates the conditional probability of the eviction of the cache line A, given the execution of a particular basic block.

Figure 5.5: An example of Ripple's eviction analysis process

of each candidate cue block. Fig. 5.5b shows an example of this probability calculation for the cache line, A. To calculate this conditional probability, Ripple calculates two metrics. First, it computes how many times each candidate cue block was executed during the application's lifetime. In this example, the candidate cue blocks B, C, D, and E are executed 16, 8, 6, and 3 times respectively. Second, for each candidate cue block, Ripple computes the number of unique eviction windows which include the corresponding candidate cue block. In our example, basic blocks B, C, D, and E are included in 4, 4, 2, and 2 unique eviction windows, respectively. Ripple calculates the conditional probability as the ratio of the second value (count of windows containing the candidate cue block) to the first value (execution count of the cue block). For instance, $P(($Eviction, $A)|($Execute, $B)) = 0.25$ denotes that for each execution of B, there is a 25% chance that the cache line A may be evicted.

Finally, for each eviction window, Ripple selects the cue block with the highest conditional probability, breaking ties arbitrarily. In our example, Ripple will select basic blocks C and E as cue blocks for 4 (windows 1, 4, 5, 6) and 2 (windows 2, 3) eviction windows, respectively. If the conditional probability of the selected basic block is larger than a threshold, Ripple will inject an explicit invalidation request in the basic block during recompilation. Next, we describe the process by which the invalidation instructions are injected as well as the trade-off that is associated with this probability threshold.

5.3.3 Injection of Invalidation Instructions

Based on the eviction analysis, Ripple selects the cue basic block for each eviction window. Next, Ripple inserts an explicit invalidation instruction into the cue block to invalidate the victim cache line. Ripple's decision to insert an invalidation instruction is informed by the conditional probability it computes for each candidate cue block. Specifically, Ripple inserts an invalidation instruction into the cue block only if the conditional probability is higher than the *invalidation threshold*. We now describe how Ripple determines the invalidation threshold and the invalidation granularity (*i.e.*, why Ripple decides to inject invalidation instructions in a basic block to evict a cache line). We then give details on the invalidation instruction that Ripple relies on.

Determining the invalidation threshold. Ripple considers two key metrics when selecting the value of the invalidation threshold: *replacement coverage* and *replacement accuracy*. We first define these metrics and then explain the trade-off between them.

Replacement-Coverage. We define *replacement-coverage* as the ratio of the total number of replacement decisions performed by a given policy divided by the total number of replacement decisions performed by the ideal replacement policy. A policy that exhibits less than 100% replacement-coverage omits some invalidation candidates that the optimal replacement policy

would have chosen for eviction.

Replacement-Accuracy. We define *replacement-accuracy* as the ratio of total optimal replacement decisions of a given policy divided by the replacement decisions performed by the ideal replacement policy. Therefore, if Ripple induces x invalidations over a program's lifetime, and y of those invalidations do not introduce any new misses over the ideal cache replacement policy, then Ripple's accuracy (in percentage) is: $\frac{100*y}{x}$. A policy that exhibits less than 100% replacement-accuracy will evict cache lines that the ideal cache replacement policy would not have evicted.

Coverage-Accuracy Trade-off. Replacement-coverage and replacement-accuracy represent useful metrics to measure a cache replacement policy's optimality. A software-guided policy with a low replacement-coverage will frequently need to revert to the underlying hardware policy suffering from its sub-optimal decisions. On the other hand, a policy with low replacement-accuracy will frequently evict lines that the program could still use. As shown in Fig. 5.6, Ripple leverages the invalidation-threshold to control the aggressiveness of its evictions, allowing to trade-off coverage and accuracy. Although this figure presents data from a single application (*i.e.*, `finagle-http`), we observe similar trends across all the data center applications that we evaluate.

At a lower threshold (0-20%), Ripple has almost 100% coverage, because all the replacement decisions are made by Ripple's invalidations. At the same time, Ripple's accuracy suffers greatly because it invalidates many cache lines that introduce new misses over the ideal cache replacement policy. Consequently, at a lower threshold, Ripple does not provide additional performance over the underlying replacement policy.

Similarly, at a higher threshold (80-100%), Ripple achieves near-perfect accuracy as cache lines invalidated by Ripple do not incur extra misses over the ideal replacement policy. However, Ripple's coverage drops sharply as more replacement decisions are not served by Ripple-inserted invalidations. Therefore, Ripple's performance benefit over the underlying hardware replacement policy declines rapidly.

Only at the middle ground, *i.e.*, when the invalidation threshold ranges from 40-60%, Ripple simultaneously achieves both high coverage (greater than 50%) and high accuracy (greater than 80%). As a result, Ripple provides the highest performance benefit at this invalidation threshold range. For each application, Ripple chooses the invalidation threshold that provides the best performance for a given application. Across 9 applications, this invalidation threshold varies from 45-65%.

Invalidation granularity. Ripple injects invalidation instructions at the basic block granularity while invalidation instructions evict cache lines. In practice, we find that Ripple does not suffer a performance loss due to this mismatch. In particular, Ripple provides a higher speedup when evicting at the basic block granularity than when evicting at the cache line or combination of basic block and cache line granularity.

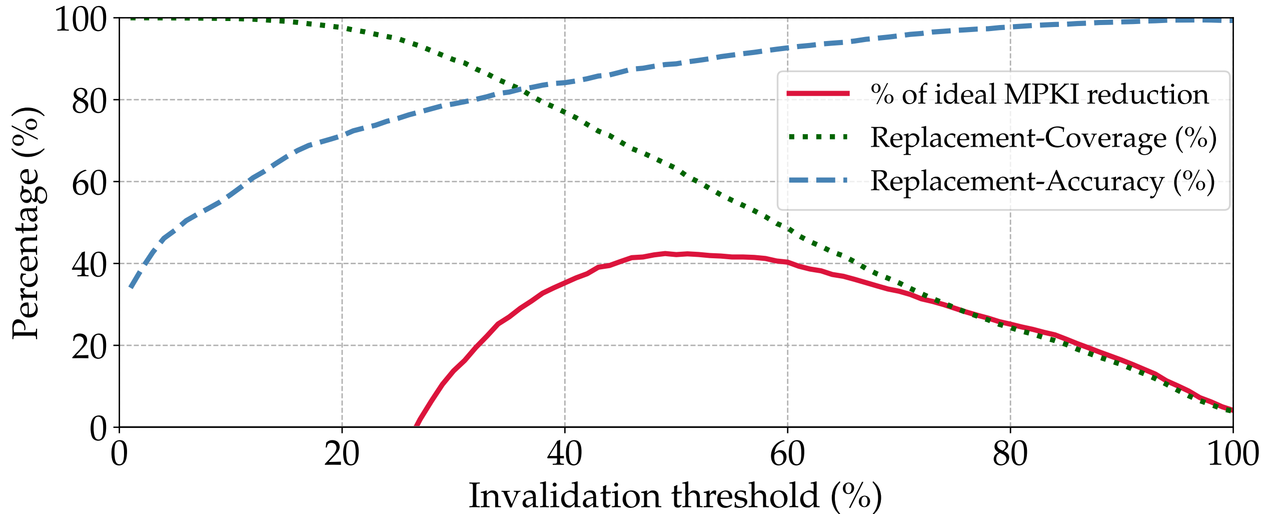


Figure 5.6: Coverage vs. accuracy trade-off of Ripple for `finagle-http`. Other applications also exhibit a similar trade-off curve. The invalidation threshold providing the best performance across the 9 data center applications we studied varies between 45-65%.

The Invalidation instruction. We propose a new invalidation instruction, `invalidate` that takes the address of a cache line as an operand and invalidates it if the cache line resides in the I-cache. Our proposed `invalidate` instruction exhibits one key difference compared to existing cache line flushing instructions (*e.g.*, the `clflush` instruction on Intel processors) in that it does not invalidate the cache line from other caches in the cache hierarchy. Instead, our proposed `invalidate` instruction invalidates the cache line only in the local I-cache, thereby avoiding costly cache-coherency transactions and unnecessary invalidations in remote caches. Furthermore, the `invalidate` instruction has low latency as it does not have to wait for the potentially dirty cache line to be written back to the lower cache levels. Instead, `invalidate` can be regarded as a hint that can be freely reordered with fences and synchronization instructions. Intel recently introduced [352] such an invalidation instruction called (`cldemote`) slated to be supported in its future servers, and hence Ripple will be readily implementable on such upcoming processors.

5.4 Evaluation

In this section, we first describe our experimental methodology and then evaluate Ripple using key performance metrics.

Trace collection. We collect the execution trace of data center applications using Intel Processor Trace (PT). Specifically, we record traces for 100 million instructions in the application's steady-state containing both user and kernel mode instructions as Intel PT allows to capture both. We find that for most applications, the percentage of kernel mode instruction induced I-cache misses

Table 5.2: Simulator Parameters

Parameter	Value
CPU	Intel Xeon Haswell
Number of cores per socket	20
L1 instruction cache	32 KiB, 8-way
L1 data cache	32 KiB, 8-way
L2 unified cache	1 MB, 16-way
L3 unified cache	Shared 10 MiB per socket, 20-way
All-core turbo frequency	2.5 GHz
L1 I-cache latency	3 cycles
L1 D-cache latency	4 cycles
L2 cache latency	12 cycles
L3 cache latency	36 cycles
Memory latency	260 cycles
Memory bandwidth	6.25 GB/s

is small ($< 1\%$). However, for `drupal`, `mediawiki`, and `wordpress`, kernel code is responsible for 15% of all I-cache misses.

Simulation. At the time of this writing, no commercially-available processor supports our proposed `invalidate` instruction, even though future Intel processors will support the functionally-equivalent `cldemote` instruction [376]. To simulate this invalidate instruction, we evaluate Ripple using simulation. This also allows us to evaluate additional replacement policies and their interactions with Ripple. We extend the ZSim simulator [310] by implementing our proposed `invalidate` instruction. We list several important parameters of the trace-driven out-of-order ZSim simulation in Table 5.2. We implement Ripple on the L1 I-cache in our experiments.

Data center applications and inputs. We use nine widely-used data center applications described in §5.2 to evaluate Ripple. We study these applications with different input parameters offered to the client's load generator (*e.g.*, number of requests per second or the number of threads). We evaluate Ripple using different inputs for training (profile collection) and evaluation.

We now evaluate Ripple using key performance metrics on all nine data center applications described in Sec. 5.2. First, we measure how much speedup Ripple provides compared to ideal and other prior cache replacement policies. Next, we compare L1 I-cache MPKI reduction (%) for Ripple, ideal, and other policies for different prefetching configurations. Then, we evaluate Ripple's replacement-coverage and replacement-accuracy as described in Sec. 5.3.3. Next, we measure how much extra static and dynamic instructions Ripple introduces into the application binary. Finally, we evaluate how Ripple performs across multiple application inputs.

Speedup. We measure the speedup (*i.e.*, percentage improvement in instructions per cycle [IPC]) provided by Ripple over an LRU baseline. We also compare Ripple's speedup to speedups provided by the prefetch-aware ideal replacement policy as well as four additional prior cache replacement

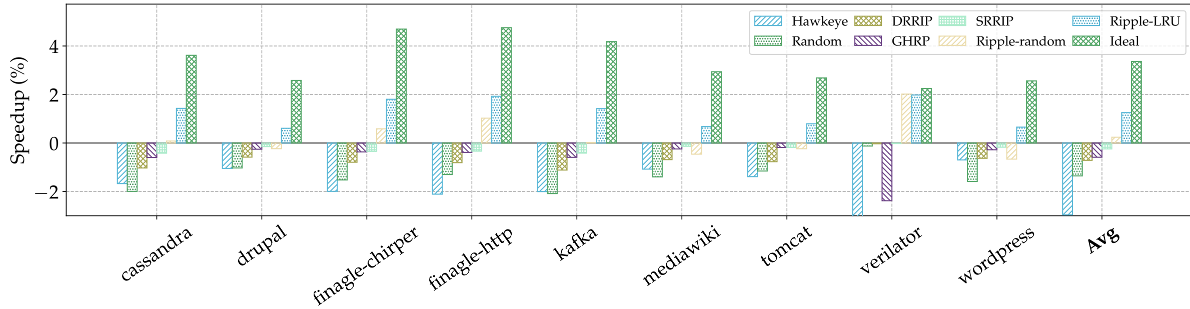
policies including Hawkeye/Harmony, DRRIP, SRRIP, and GHRP, whose details were discussed in §5.2. To show that Ripple's speedup is not primarily due to the underlying hardware replacement policy, we also provide Ripple's speedup with two different underlying hardware replacement policies (random and LRU). Finally, we measure the speedups for all replacement policies by altering the underlying I-cache prefetching mechanisms (no prefetching, NLP, and FDIP).

Fig. 5.7 shows the speedup results. Ripple, with an underlying LRU-based hardware replacement policy (*i.e.*, Ripple-LRU in Fig. 5.7), always outperforms all prior replacement policies across all different prefetcher configurations. In particular, Ripple-LRU provides on average 1.25% (no prefetching), 2.13% (NLP), 1.4% (FDIP) speedups over a pure-LRU replacement policy baseline. These speedups correspond to 37% (no prefetching), 55% (NLP), and 44% (FDIP) of the speedups of an ideal cache replacement policy. Notably, even Ripple-Random, which operates with an underlying random hardware replacement policy (which itself is on average 1% slower than LRU), provides 0.86% average speedup over the LRU baseline across the three different I-cache prefetchers. In combination with Ripple, Random becomes a feasible replacement policy that eliminates all meta-data storage overheads in hardware.

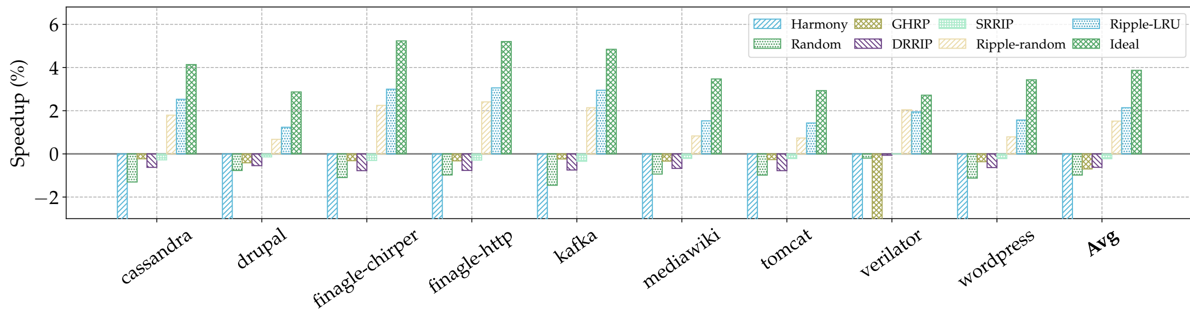
The performance gap between Ripple and the ideal cache replacement policy stems from two primary reasons. First, Ripple cannot cover all eviction windows via software invalidation as covering all eviction windows requires Ripple to sacrifice eviction accuracy which hurts performance. Second, software invalidation instructions inserted by Ripple introduce static and dynamic code bloat, causing additional cache pressure that contributes to the performance gap (we quantify this overhead later in this section).

I-cache MPKI reduction. Fig. 5.8 shows the L1 I-cache miss reduction provided by Ripple (with underlying hardware replacement policies of LRU and Random) and the prior work policies. As shown, Ripple-LRU reduces I-cache misses over all prior policies across all applications. Across different prefetching configurations, Ripple can avoid 33% (no prefetching), 53% (NLP), and 41% (FDIP) of I-cache misses that are avoided by the ideal replacement policy. Ripple reduces I-cache MPKI regardless of the underlying replacement policy. Even when the underlying replacement policy is random (causing 12.71% more misses in average than LRU), Ripple-Random incurs 9.5% fewer misses on average than LRU for different applications and prefetching configurations.

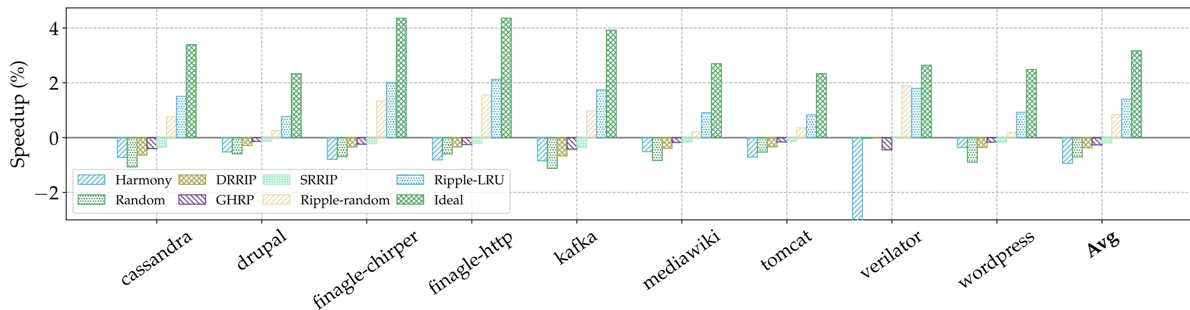
Replacement-Coverage. As described in §5.3.3, Ripple's coverage is the percentage of all replacement decisions (over the program life time) that were initiated by Ripple's invalidations. Fig. 5.9 shows Ripple's coverage for all applications. As shown, Ripple achieves on average more than 50% coverage. Only for three `HHVM` applications (*i.e.*, `drupal`, `mediawiki`, and `wordpress`), Ripple's coverage is lower than 50%, as for these applications Ripple does not insert `invalidate` instructions on the just-in-time compiled basic blocks. Just-in-time (Jit) compiled code may reuse the same instruction addresses for different basic blocks over the course of an



(a) Over no prefetching baseline, Ripple provides 1.25% speedup compared to 3.36% ideal speedup on average.

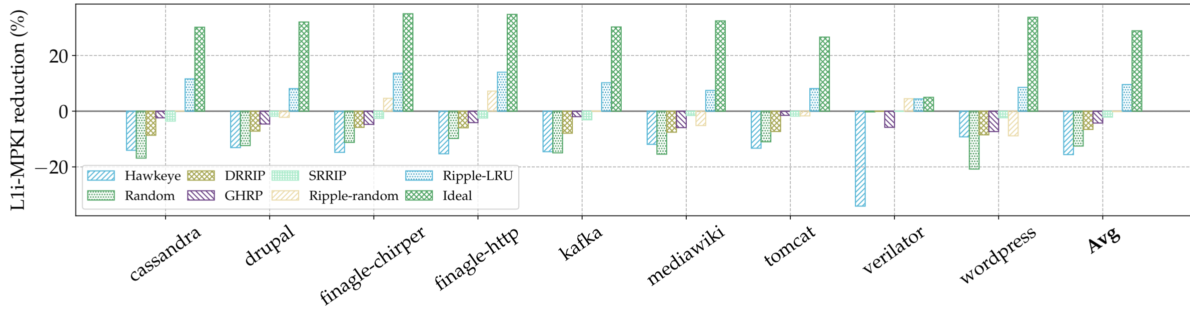


(b) Over next-line prefetching baseline, Ripple provides 2.13% speedup compared to 3.87% ideal speedup on average.

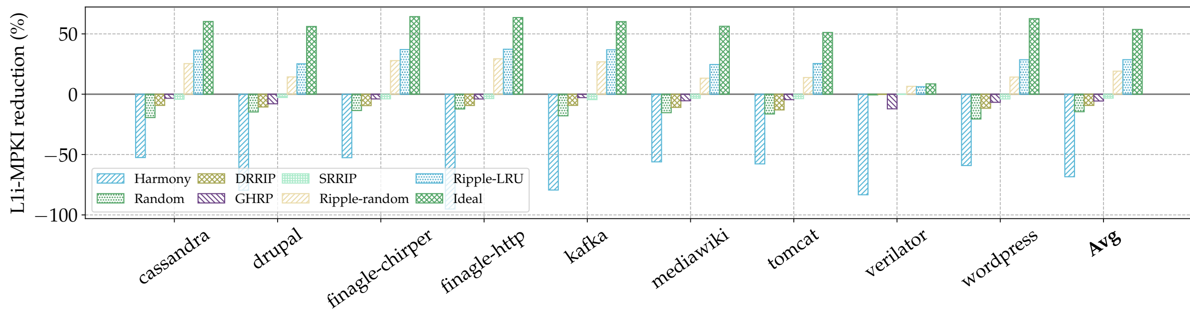


(c) Over fetch directed instruction prefetching baseline, Ripple provides 1.4% speedup compared to 3.16% ideal speedup on average.

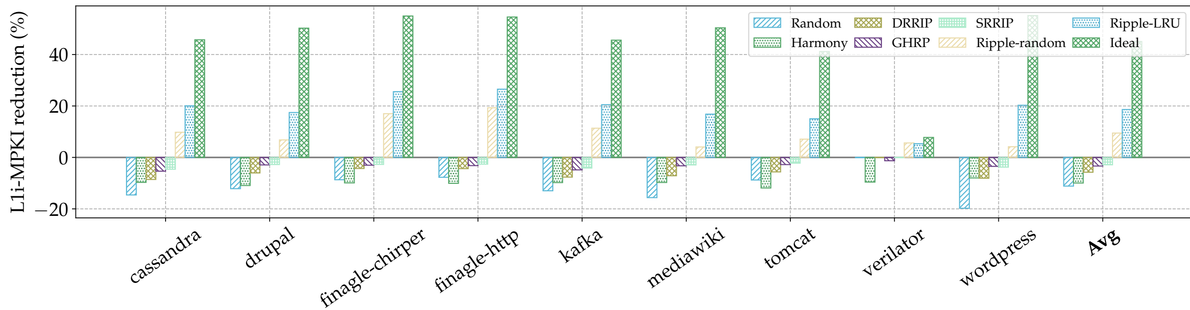
Figure 5.7: Ripple's speedup compared to ideal and state-of-the-art replacement policies over an LRU baseline (with different hardware prefetching): On average, Ripple provides 1.6% speedup compared to 3.47% ideal speedup.



(a) With no prefetching, Ripple-LRU reduces 9.57% of all I-cache misses compared to 28.88% miss reduction provided by the ideal replacement policy.



(b) With next-line prefetching, Ripple-LRU reduces on average 28.6% of all I-cache misses compared to 53.66% reduction provided by the ideal replacement policy.



(c) With fetch directed instruction prefetching, Ripple-LRU reduces on average 18.61% of all I-cache misses compared to 45% reduction provided by the ideal replacement policy.

Figure 5.8: Ripple's L1 I-cache miss reduction compared to ideal and state-of-the-art replacement policies over an LRU baseline (with different hardware prefetching): On average, Ripple reduces 19% of all LRU I-cache misses compared to 42.5% miss reduction by the ideal replacement policy.

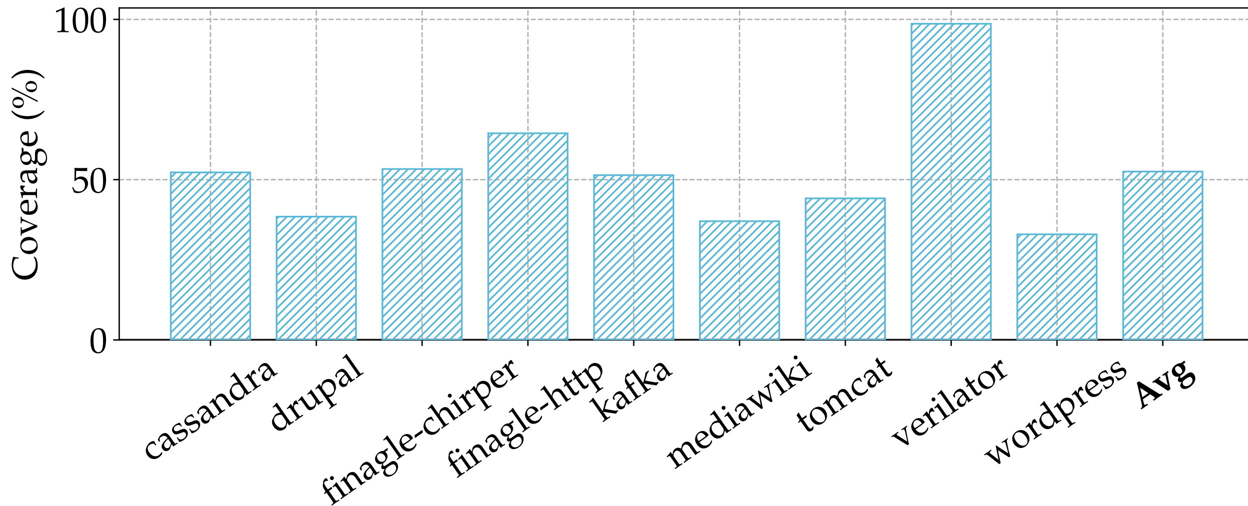


Figure 5.9: Ripple's coverage for different applications: On average 50% of replacement requests are processed by evicting cache lines that Ripple invalidates.

execution rendering compile-time instruction injection techniques challenging. Nevertheless, even for these Jit applications there remains enough static code that Ripple is able to optimize.

Accuracy. In Fig. 5.10, we show Ripple's replacement-accuracy (as defined in §5.3.3). As shown, Ripple achieves 92% accuracy on average (with a minimum improvement of 88%). Ripple's accuracy is on average 14% higher than LRU's average accuracy (77.8%). Thanks to its higher accuracy, Ripple avoids many inaccurate replacement decisions due to the underlying LRU-based hardware replacement policy (which has an average accuracy of 77.8%), and, therefore, the overall replacement accuracy for Ripple-LRU is on average 86% (8.2% higher than the LRU baseline).

Instruction overhead. Fig. 5.11 and 5.12 quantify the static and dynamic code footprint increase introduced by the injected invalidate instructions. The static instruction overhead of Ripple is less than 4.4% for all cases while the dynamic instruction overhead is less than 2% in most cases, except for `verilator`. For this application, Ripple executes 10% extra instructions to invalidate cache lines. This is because for `verilator`, Ripple covers almost all replacement policy decisions via software invalidation (98.7% coverage as shown in Fig. 5.9). Similarly, Ripple's accuracy for `verilator` is very high (99.9% as shown in Fig. 5.10). Therefore, though Ripple executes a relatively greater number of invalidation instructions for `verilator`, it does not execute unnecessary invalidation instructions.

Profiling and offline analysis overhead. Ripple leverages Intel PT to collect basic block traces from data center application executions because of its low overhead (less than 1%) and adoption in production settings [86, 121]. While Ripple's extraction and analysis on this trace takes longer (up to 10 minutes), we do not expect that this expensive analysis will be deployed in production servers. Instead, we anticipate the extraction, analysis, and invalidation injection component of

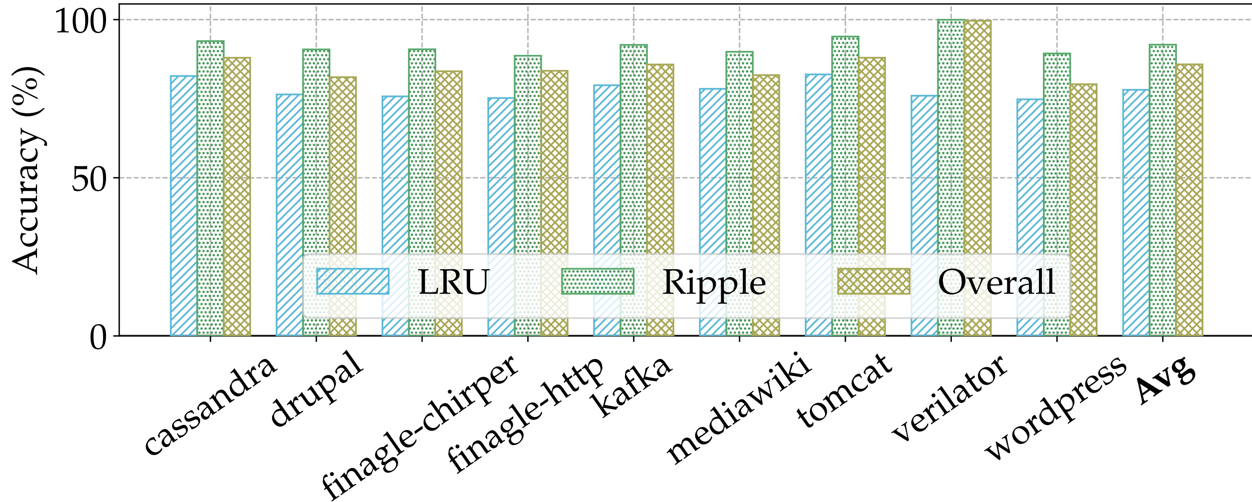


Figure 5.10: Ripple's accuracy for different applications: On average Ripple provides 92% accuracy which ensures that the overall accuracy is 86% even though underlying LRU has an accuracy of 77.8%.

Ripple will be performed offline, similar to how existing profile-guided optimizations for data center applications are performed [65, 278, 126, 197, 279]. Therefore, we consider the overhead for Ripple's offline analysis acceptable.

Invalidation vs. reducing LRU priority. When the underlying hardware cache replacement policy is LRU, moving a cache line to the bottom of the LRU chain is sufficient to cause eviction. This LRU-specific optimization improved Ripple's IPC speedup from 1.6% to 1.7% (apart from `verilator`, all other applications benefited from this optimization). This shows that Ripple's profiling mechanism works well independent of the particular eviction mechanism.

Performance across multiple application inputs. We investigate Ripple's performance for data center applications with three separate input configurations ('`#1`' to '`#3`'). We vary these applications' input configurations by changing the webpage, the client requests, the number of client requests per second, the number of server threads, random number seeds, and the size of input data. We optimize each application using the profile from input '`#0`' and measure Ripple's performance benefits for different test inputs '`#1, #2, #3`'. For each input, we also measure the performance improvement when Ripple optimizes the application with a profile for the same input. As shown in Fig. 5.13, Ripple provides 17% more IPC gains with input-specific profiles compared to profiles that are not input specific. For brevity, we only show the results for the FDIP baseline. Results with the other prefetching baselines are similar.

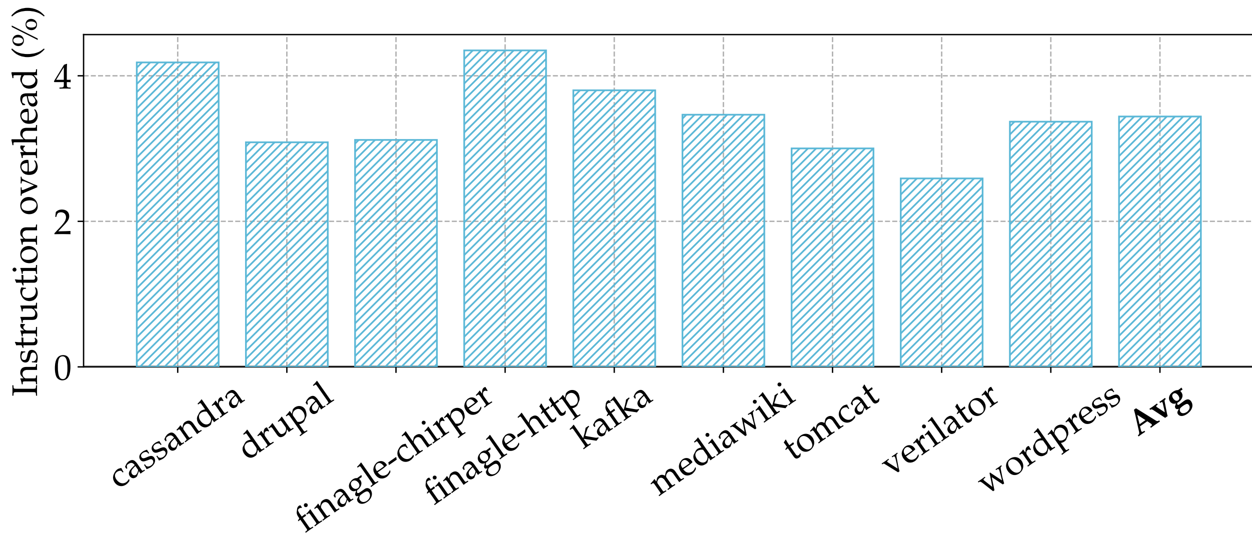


Figure 5.11: Static instruction overhead introduced by Ripple: On average Ripple inserts 3.4% new static instructions.

5.5 Discussion

Ripple generates optimized binaries for different target architectures considering the processor's I-cache size and associativity. Such a process is common in data centers deploying profile-guided [65, 216] and post-link-time-based optimization [278, 279] techniques. Therefore, Ripple can be conveniently integrated into the existing build and optimization processes. Moreover, as the I-cache size (32KB) and associativity (8-way) for Intel data center processors has been stable for the last 10 years, the number of different target architectures that Ripple needs to support is small.

5.6 Related Work

Instruction prefetching. Hardware instruction prefetchers such as next-line and decoupled fetch directed prefetchers [328, 306, 152, 68, 302] have been pervasively deployed in commercial designs [350, 283, 308, 129]. While complex techniques [114, 113, 189, 190] employing record and replay prefetchers are highly effective in reducing I-cache misses, they require impractical on-chip metadata storage. Branch predictor-guided prefetchers [211, 212, 34], on the other hand, follow the same principle as FDIP to reduce on-chip metadata storage, however, they also require a complete overhaul of the underlying branch target prediction unit. Even recent proposals [268, 134, 33, 320, 254, 305, 128, 122] from 1st Instruction Prefetching Championship (IPC1) require kilobytes of extra on-chip storage to provide near-ideal performance even on workloads where FDIP with a large enough fetch target queue provides most of the potential performance

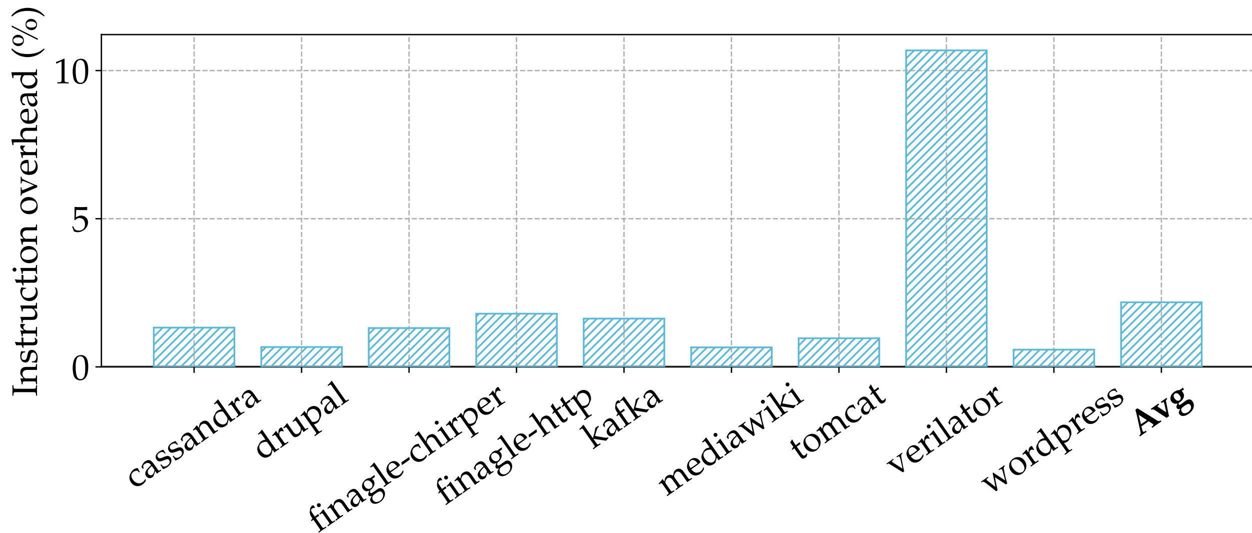


Figure 5.12: Dynamic instruction overhead introduced by Ripple: On average Ripple executes 2.2% extra dynamic instructions.

benefit [150]. Hybrid hardware-software prefetchers [41, 266, 40, 238, 199] analyze a program's control flow information in software and inject dedicated prefetching instructions in code which does not exist in today's hardware. In contrast, we show that instruction prefetchers alone do not close the performance gap due to wasteful evictions that must be handled by smarter cache line replacement.

Cache replacement policies. Heuristic-based hardware data cache replacement policies have been studied for a long time, including LRU and its variations [182, 219, 273, 327, 387], MRU [297], re-reference interval prediction [156], reuse prediction [100, 108, 227] and others [21, 118, 144, 205, 298, 314, 349, 355]. Learning-based data cache replacement policies [153, 154, 193, 390] consider replacement as a binary classification problem of cache-friendly or cache-averse. Recent methods introduce machine learning techniques like perceptrons [168, 359] and genetic algorithms [161]. Some learning-based policies use information of Belady's optimal solution [48], including Hawkeye [153], Glider [326] and Parrot [226]. However, these policies are mostly designed for data caches and do not work well for instruction caches as we show earlier (Sec. 5.2). We also propose a profile-guided approach that can work on top of any of these policies.

Prefetch-aware replacement policy. Prefetch-aware replacement policies focus on avoiding cache pollution caused by inaccurate prefetches. Some prefetch-aware policies [148, 149, 206] get feedback from prefetchers to identify inaccurate prefetches, and need co-design or prefetcher modifications. Others [341, 315, 391] work independently from the prefetcher and estimate prefetch accuracy from cache behavior.

With prefetching, Belady's optimal policy [48] becomes incomplete as it cannot distinguish

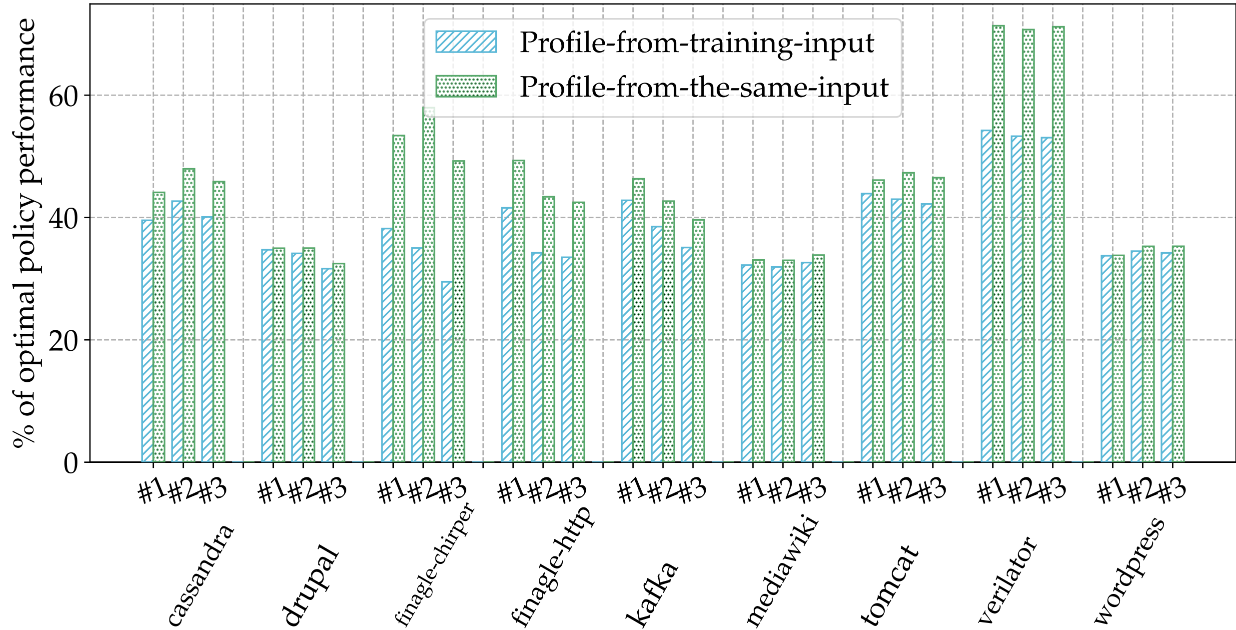


Figure 5.13: Ripple's performance for multiple application inputs with the FDIP baseline: On average Ripple provides 17% more speedup with input-specific profiles compared to profiles from different inputs.

easy-to-prefetch cache lines from hard-to-prefetch cache lines [341, 391]. To address this limitation, Demand-MIN [154] revised Belady's optimal policy to accommodate prefetching and proposed a program counter (PC) classification based predictor, Harmony to emulate the ideal performance. In this work, we not only revise Demand-MIN to cover an extra corner case, but also show that a PC-classification based predictor performs poorly for I-cache. We address this imprecision and effectively emulate optimal I-cache behavior in our work via a profile-guided software technique.

5.7 Conclusion

Modern data center applications have large instruction footprints, leading to significant I-cache misses. Although numerous prior proposals aim to mitigate I-cache misses, they still fall short of an ideal cache. We investigated why existing I-cache miss mitigation mechanisms achieve sub-optimal speedup, and found that widely-studied instruction prefetchers incur wasteful prefetch-induced evictions that existing replacement policies do not mitigate. To enable smarter evictions, we proposed Ripple, a novel profile-guided replacement technique that uses program context to inform the underlying replacement policy about efficient replacement decisions. Ripple identifies program contexts that lead to I-cache misses and sparingly injects "cache line eviction" instructions

in suitable program locations at link time. We evaluated Ripple using nine popular data center applications and demonstrated that it is replacement policy agnostic, *i.e.*, it enables any replacement policy to achieve speedup that is 44% closer to that of an ideal I-cache.

CHAPTER 6

Twig: Profile-Guided BTB Prefetching for Data Center Applications

Modern data center applications have deep software stacks, with instruction footprints that are orders of magnitude larger than typical instruction cache (I-cache) sizes. To efficiently prefetch instructions into the I-cache despite large application footprints, modern server-class processors implement a decoupled frontend with Fetch Directed Instruction Prefetching (FDIP). In this work, we[1] first characterize the limitations of a decoupled frontend processor with FDIP and find that FDIP suffers from significant Branch Target Buffer (BTB) misses. We also find that existing techniques (*e.g.*, stream prefetchers and predecoders) are unable to mitigate these misses, as they rely on an incomplete understanding of a program's branching behavior.

To address the shortcomings of existing BTB prefetching techniques, we propose Twig, a novel profile-guided BTB prefetching mechanism. Twig analyzes a production binary's execution profile to identify critical BTB misses and inject BTB prefetch instructions into code. Additionally, Twig coalesces multiple non-contiguous BTB prefetches to improve the BTB's locality. Twig exposes these techniques via new BTB prefetch instructions. Since Twig prefetches BTB entries without modifying the underlying BTB organization, it is easy to adopt in modern processors. We study Twig's behavior across nine widely-used data center applications, and demonstrate that it achieves an average 20.86% (up to 145%) performance speedup over a baseline 8K-entry BTB, outperforming the state-of-the-art BTB prefetch mechanism by 19.82% (on average).

6.1 Introduction

Modern data center applications have deep software stacks that are composed of complex application logic [275], diverse libraries [179], and numerous kernel modules [41, 211, 212]. Such deep

[1] Some of the work in this chapter was performed in collaboration with Nathan Brown, Akshitha Sriraman, Niranjan Soundararajan, Rakesh Kumar, Joseph Devietti, Sreenivas Subramoney, Gilles Pokam, Heiner Litz, and Baris Kasikci [194]. Therefore, I use the "we" pronoun in this chapter to acknowledge their involvement in this work.

stacks result in multi-megabyte instruction footprints [179, 41, 279] that easily exhaust typical on-chip cache structures which are smaller than hundred kilobytes [39]. As a result, data center applications suffer from significant frontend stalls, when the processor frontend is unable to supply instructions to the processor backend. Such frontend stalls significantly hurt the Total Cost of Operation of a data center, as even single-digit performance improvements of frontend stalls can save millions of dollars and meaningfully reduce the global carbon footprint [344].

Processor architects attempt to address this overwhelming frontend stall problem by proposing numerous instruction prefetching mechanisms [328, 302, 114, 280, 113, 189, 209, 212, 211]. Fetch Directed Instruction Prefetching (FDIP) [302] is one such mechanism that is pervasively explored in academia [212, 211, 202] and industry [150, 151]. Between the branch prediction unit and the instruction fetch engine, FDIP introduces a queue containing the addresses of I-cache lines that will be accessed in the future [301]. FDIP prefetches I-cache lines based on the queue contents to avoid instruction fetch stalls. FDIP allows the branch prediction unit and the instruction fetch engine to operate independently with high efficiency. Prior work [150] has shown that FDIP provides comparable performance to aggressive I-cache prefetchers [320, 268, 305] used in recent instruction prefetching championships. Due to its success, FDIP has been widely implemented in modern processors [350, 283, 308, 129].

Given that data center applications still continue to face the frontend stall problem, we first ask the question: What limits FDIP from eliminating all frontend stalls? To this end, we comprehensively study FDIP in the context of frontend-bound data center applications and show that FDIP still falls significantly short of an ideal I-cache (by 24% on average). We also find that FDIP's effectiveness primarily depends on the efficacy of the Branch Target Buffer (BTB); therefore, the large number of BTB misses, which is typical for data center applications, hurts FDIP's effectiveness. We then investigate the reasons behind the large number of BTB misses for data center applications. We find that these applications contain a large number of unique branch instructions that cannot fit into moderately-sized BTBs. Furthermore, we show that the state-of-the-art BTB prefetching techniques, such as Shotgun [211] and Confluence [190], suffer from limited prefetching coverage and accuracy while introducing significant hardware modifications. For this reason, they have not been adopted in modern data center processors [41, 199].

In this paper, we propose Twig, a novel profile-guided BTB prefetching mechanism for data center applications. Unlike prior techniques [190, 211], Twig does not require any modifications to the typical BTB organization. Instead, Twig introduces a new BTB prefetching instruction that is directly injected into the program binary at link time. By inserting BTB prefetch instructions in software, Twig leverages the rich execution information available in a program profile, when collected using performance counters in modern data center environments [65, 278, 179, 41].

Twig introduces two key techniques: *software BTB prefetching* and *BTB prefetch coalescing*.

Software BTB prefetching. A BTB entry is composed of a branch instruction address and a corresponding branch target address. To prefetch a BTB entry, the processor has to decode the branch target of a given branch instruction. However, the branch instruction itself may not be present in the I-cache, rendering BTB prefetching impossible. Twig addresses this challenge by introducing an explicit *prefetch* instruction to prefetch BTB entries in advance, without bringing the required instructions into the I-cache. This *prefetch* instruction prefetches branch instruction address and target into the BTB. Unlike pure hardware techniques that rely on limited past run-time information [190, 211], Twig determines which branch instructions cause frequent BTB misses based on profiles collected from the entire program execution. Twig's prefetch instruction takes as operands the address of the branch instruction and the address of the corresponding target instruction. Twig then ensures that the corresponding entry is inserted into the BTB even if the branch instruction is not in the I-cache.

Twig further leverages production execution profiles to identify program locations that can predict the future execution of a BTB-miss inducing branch instruction with high accuracy and timeliness. Twig then inserts prefetch instructions into these locations.

BTB prefetch coalescing. Inserting many BTB prefetch instructions with multiple parameters can increase the static and dynamic instruction footprint. To mitigate this code bloat, Twig proposes *BTB prefetch coalescing*, where multiple BTB entries are prefetched with a single instruction. Twig analyzes the program profile to identify consecutively-executed branches that incur repetitive BTB misses. Consequently, Twig uses the coalesced prefetching instruction to prefetch the BTB entries of all of these branch instructions simultaneously.

We evaluate Twig in the context of nine data center applications that suffer from frequent frontend stalls. Twig achieves an average 20.86% (2%-145%) speedup over a baseline 8K-entry BTB across all nine applications, while reducing 65.4% of all BTB misses. Compared to the state-of-the-art BTB prefetcher [211], Twig achieves an average 19.82% (up to 139.8%) greater speedup, while covering 57.4% more BTB misses. Twig's average static and dynamic instruction increase overhead is 6% and 3% respectively.

In summary, we contribute:

- A detailed characterization of a decoupled frontend with FDIP that shows that a large number of BTB misses hurt FDIP's effectiveness.
- *Software BTB prefetching:* A technique to prefetch BTB entries that improves the decoupled frontend's performance by avoiding costly BTB misses.
- *BTB prefetch coalescing:* A profile-guided mechanism to coalesce multiple BTB prefetch operations that reduces prefetch instructions' static and dynamic overhead.
- An evaluation of Twig in the context of nine data center applications, showing its effectiveness in reducing BTB misses and achieving significant performance benefit.

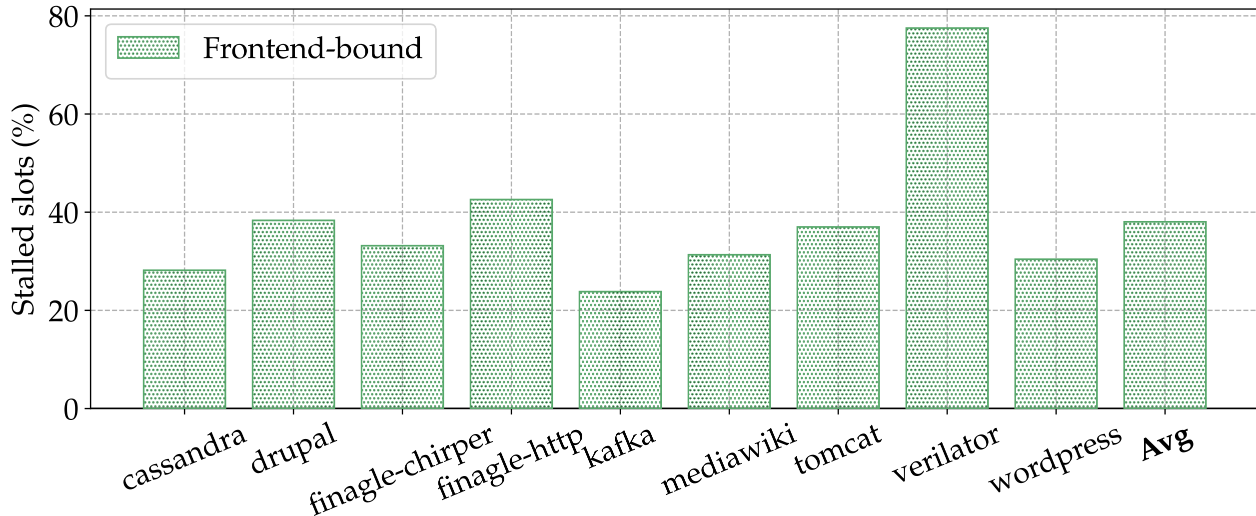


Figure 6.1: Many popular data center applications waste a large portion of their pipeline slots due to "frontend-bound" stalls [147], measured using the Top-down methodology [395].

6.2 Limitations of prior I-Cache & BTB Prefetching Techniques

In this section, we comprehensively characterize existing I-cache and BTB prefetching mechanisms to understand why data center applications continue to suffer from frontend stalls. We first analyze FDIP [302], the state-of-the-art prefetching technique in processors with a decoupled frontend. We measure the unrealized performance potential of FDIP and find that its performance is mainly limited by BTB misses. We then analyze Shotgun [211] and Confluence [190], two recently proposed techniques that introduce BTB prefetching on top of FDIP. While these techniques reduce BTB misses for some applications, they fail to eliminate BTB misses that occur due to complex branch patterns faced by data center applications.

We characterize nine popular real-world data center applications [199] that face significant frontend stalls. In Fig. 6.1, we use Intel's Top-Down methodology [395] to show that these applications spend 24%-78% of the processor pipeline slots in waiting for the frontend to return. Two applications, `finagle-chirper` (a microblogging service) and `finagle-http` (an HTTP server) are from the Java Renaissance [294] benchmark suite and use Twitter Finagle [15] which is a Remote Procedure Call (RPC) library. Three applications, `kafka` [377] (Apache stream-processing framework used by companies like Uber, Linkedin, and Airbnb [3]), `tomcat` [4] (open-source Java web server), and `cassandra` [2] (NoSQL DBMS used by companies like Uber, Netflix, and Grubhub [384]) are from the Java DaCapo [51] benchmark suite. We also study three HHVM [23, 274] applications (`drupal`, `wordpress`, and `mediawiki`) from Facebook's

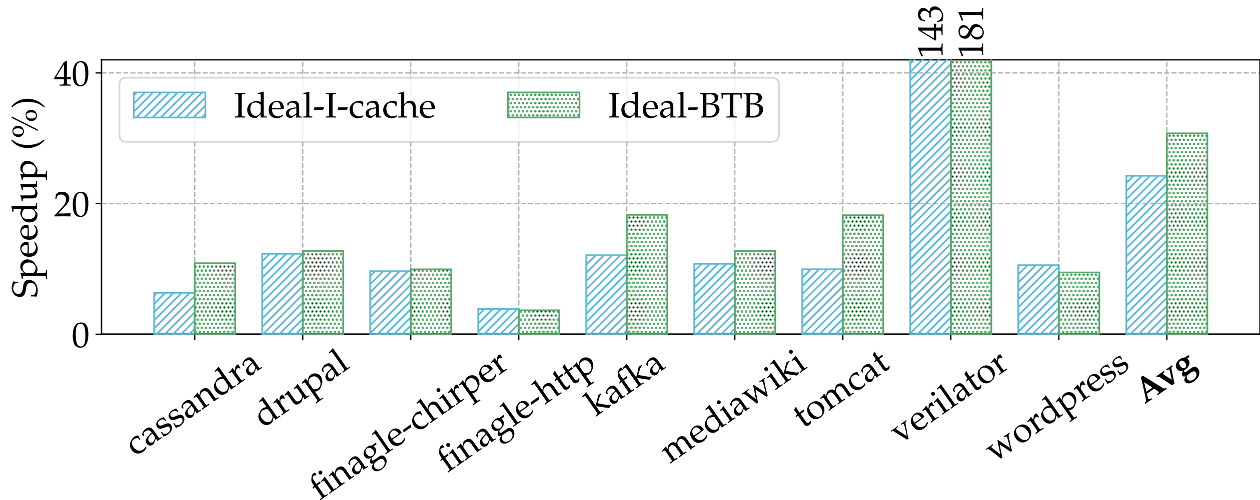


Figure 6.2: Limit study of FDIP: an ideal I-cache achieves an average 24% speedup, while an ideal BTB provides an average 31% speedup over the FDIP baseline.

OSS-performance [19] benchmark suite. `verilator` [16] is a tool used by companies like Intel and ARM to evaluate custom hardware designs [382]. We detail our experimental setup, trace collection methodology, and simulation parameters in §6.4.

6.2.1 What stops FDIP from eliminating all frontend stalls?

Recent processor designs [350, 283, 129, 308] have adopted decoupled frontends with FDIP to reduce costly frontend stalls. Given FDIP's widespread adoption [150, 151], we ask the question: Does FDIP achieve performance comparable to an ideal/perfect frontend where pipeline slots are not stalled in the frontend? To this end, we analyze FDIP's limitations, characterizing why FDIP falls short for data center applications. Additionally, we determine how to address FDIP's limitations.

We perform two limit studies, measuring the Instructions Per Cycle (IPC) metric of nine data center applications running on an FDIP-enabled processor. In the first study, we analyze FDIP with an ideal I-cache (*i.e.*, every I-cache access is a hit), and in the second study, we analyze FDIP with an ideal BTB (*i.e.*, every branch target lookup is a hit). We assume a 75KB 8K-entry BTB and a 32KB I-cache. Fig. 6.2 shows an average IPC improvement of 24% with an ideal I-cache and a 31% improvement with an ideal BTB. FDIP with an ideal BTB offers greater performance benefits since (1) it eliminates almost all I-cache misses (due to FDIP prefetching) and (2) it reduces branch resteers (*i.e.*, pipeline flushes) triggered by BTB misses. Hence, we conclude that reducing BTB misses is critical to mitigating frontend stalls. Next, we investigate why data center applications suffer from poor BTB locality even with a relatively large, 75KB 8K-entry BTB.

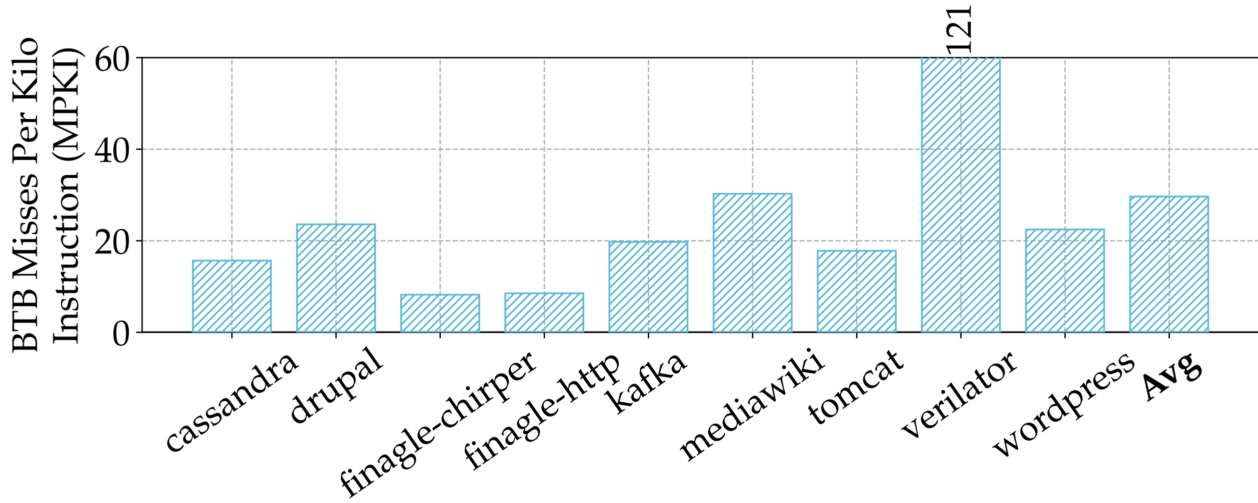


Figure 6.3: BTB Misses Per Kilo Instructions (MPKI) for nine data center applications: these applications experience an average BTB MPKI of 29.7 (8-121).

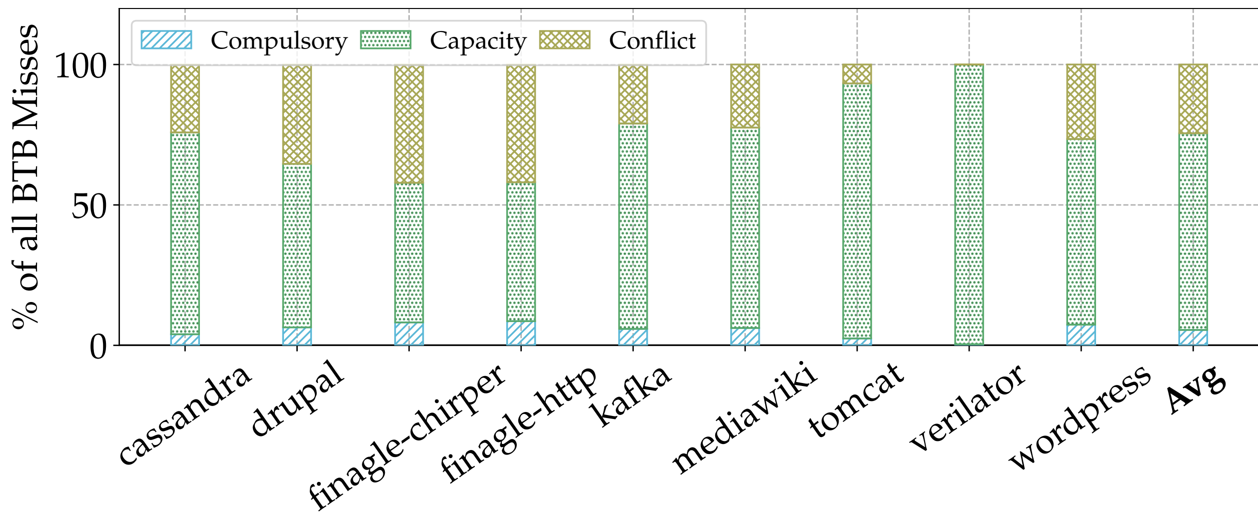


Figure 6.4: Breakdown of all BTB misses using 3C miss classification [142]: data center applications suffer BTB misses due to both capacity and conflict issues.

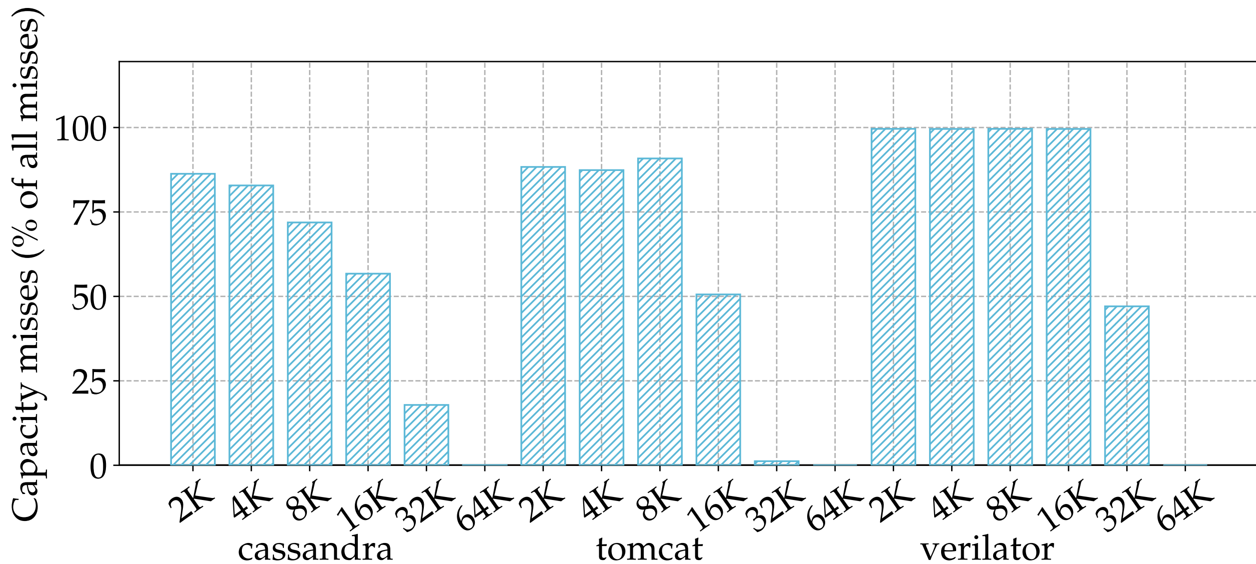


Figure 6.5: Percentage of capacity misses as BTB size increases from 2K to 64K entries: data center applications require large BTB with at least 32K entries to avoid all capacity misses. For brevity, we show results for only 3 applications, but the behavior is similar across all applications.

6.2.2 Why is a large BTB insufficient for data center applications?

As an ideal BTB significantly improves FDIP's performance, we examine how we can improve the performance of the 75KB 8K-entry BTB that is implemented in today's FDIP-enabled processors.

Fig. 6.3 shows the BTB Misses Per Kilo Instructions (MPKI) across all nine data center applications. While measuring BTB MPKI, we only consider real BTB misses caused by direct branch instructions, *i.e.*, unconditional jumps, calls, and conditional jumps. We do not include non-control flow instructions or branch instructions where the branch target that the BTB returns is different from the actually taken branch target (*e.g.*, branch target changed due to just-in-time code compilation).

As shown in Fig. 6.3, data center applications experience MPKIs in the range of 8-121 (29.7 on average). To understand the reason behind significant BTB misses, in Fig. 6.4, we categorize whether these misses are *compulsory*, *capacity*, or *conflict* misses, *i.e.*, the 3C miss classification [142]. We find that the majority of these misses are capacity (on average 70%) and conflict (on average 24.48%) misses.

To investigate these capacity and conflict misses, we vary the BTB size (from 2K entries to 64K entries) and associativity (from 4-way to 128-way) and show the results in Fig. 6.5 and Fig. 6.6. We observe that these data center applications require a 64K-entry BTB to avoid most of the capacity misses. On the other hand, the BTB associativity needs to be at least 128 to cover the majority of conflict misses. Increasing BTB size and associativity to these levels will drastically increase

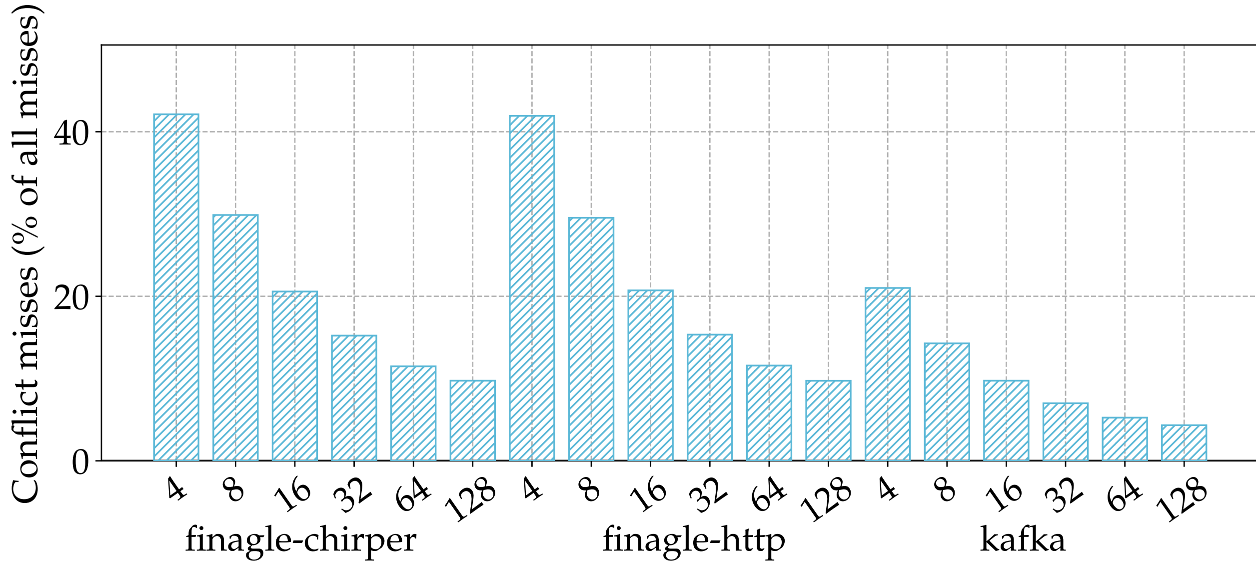


Figure 6.6: Percentage of conflict misses as BTB associativity increases from 4-way to 128-way: data center applications still suffer conflict BTB misses even with an 128-way set-associative BTB. For brevity, we show results for only 3 applications, the behavior is similar across all applications.

on-chip storage and BTB lookup/update latency [60, 165]. Furthermore, future applications may require an even larger BTB size and associativity since data center applications' instruction footprints grow in an unprecedented manner [179]. Therefore, we conclude that BTB prefetching is a more future-proof solution as it can avoid latencies due to both types of BTB misses without requiring any change to the BTB organization.

Finally, in Fig. 6.7 and Fig. 6.8, we study the distribution of all BTB accesses and misses across different branch types to identify whether a specific branch type suffers from poor BTB locality. We note that unconditional direct branches and calls disproportionately face more BTB misses. Specifically, unconditional direct branches and calls are responsible for 20.75% of all dynamic branches, but incur 37.5% of all BTB misses. This result justifies the design decisions of prior work [211] that partitions the BTB structure to prefetch conditional branch entries that follow unconditional branch executions.

6.2.3 Why do existing BTB prefetching mechanisms fall short?

Previously, we showed that an ideal BTB provides on average 31% speedup over the FDIP baseline. We now compare this ideal BTB speedup against speedups achieved by state-of-the-art BTB prefetchers, Confluence [190] and Shotgun [211].

Confluence observes that although the I-cache and the BTB operate at the granularity of a cache line and a branch instruction respectively, hardware prefetching mechanisms for I-cache

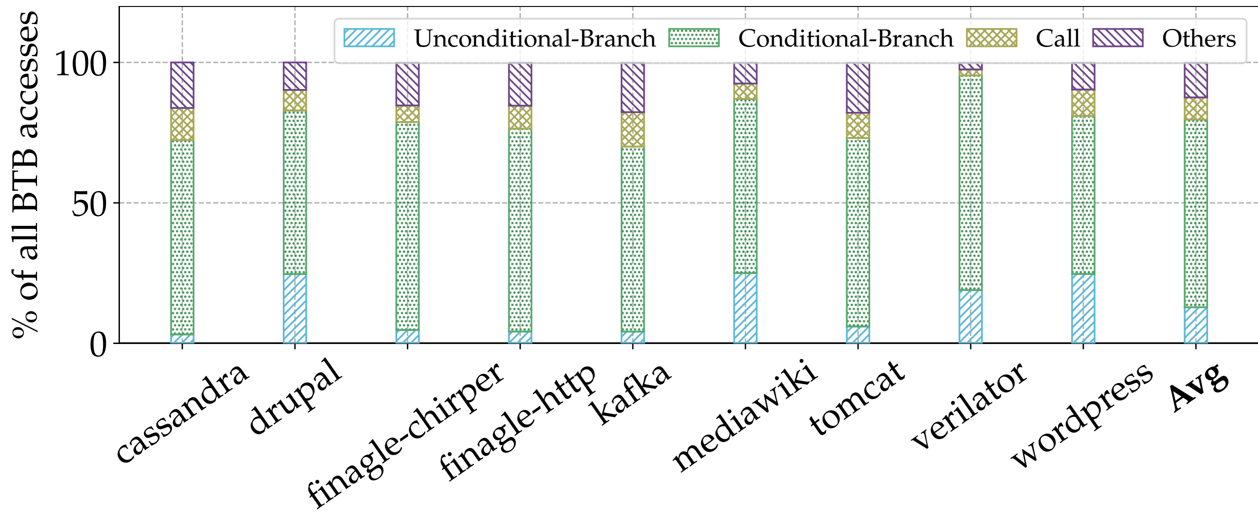


Figure 6.7: Breakdown of all BTB accesses into branch types: conditional branch instructions dominate the total number of BTB accesses

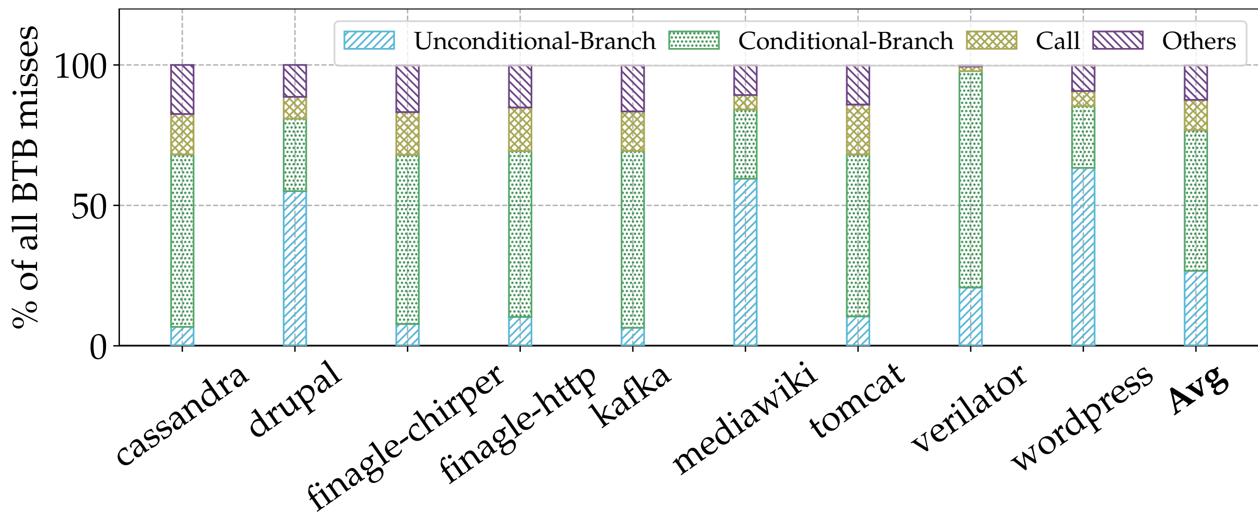


Figure 6.8: Breakdown of all BTB misses into different branch types: as conditional branch instructions are responsible for most BTB accesses, conditional branch instructions also experience the most number of BTB misses.

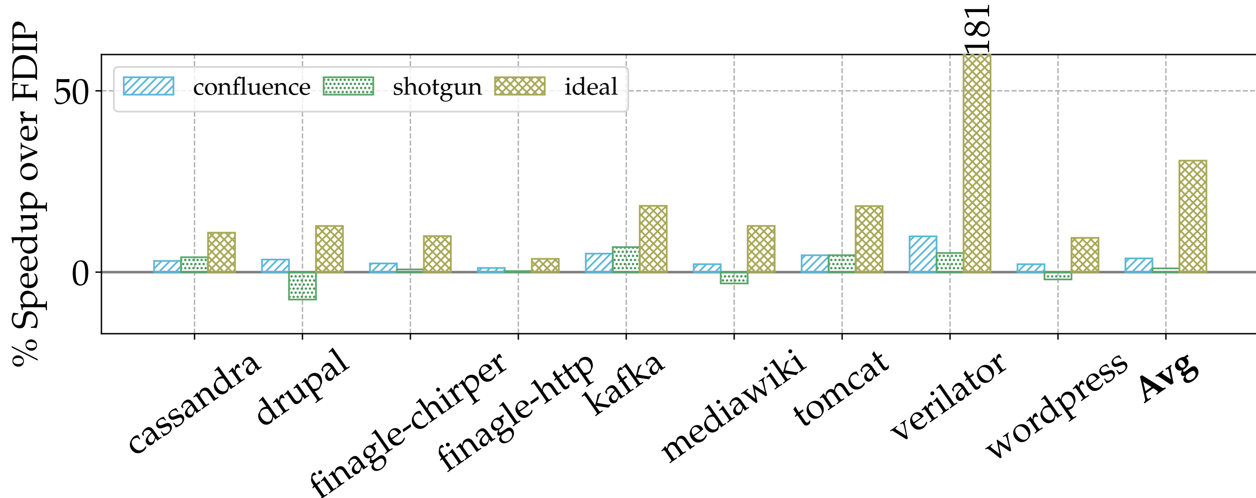


Figure 6.9: Speedups from Shotgun and Confluence over FDIP.

lines and BTB entries require the same metadata. Using this insight, Confluence (1) modifies the BTB organization to match the I-cache granularity (cache line), (2) operates on the same prefetch metadata, and (3) utilizes the temporal streaming (also referred to as "record and replay" [114, 113, 189]) technique, to perform both I-cache and BTB prefetching. While Confluence was designed for a fixed-length instruction size (4B), we modify Confluence for variable-length instruction sizes since most data center applications operate on servers that use variable-length ISAs (*i.e.*, x86).

Shotgun observes that the working set size of unconditional branch instructions is significantly smaller than the working set size of all branch instructions. Hence, Shotgun statically partitions the BTB among unconditional and conditional branch entries to ensure that a certain type of branch entry does not cause evictions of the other type. Moreover, Shotgun leverages dynamic execution information to record the I-cache footprint for all unconditional branches. The next time the program executes the same unconditional branch, Shotgun prefetches the recorded I-cache lines (if not present in the I-cache) and predecodes the corresponding conditional branch entries. In our evaluation, Shotgun consists of 5120-entry unconditional BTB (63.125KB), 1536-entry conditional BTB (12.1875KB), and 1536-entry return address stack (7.5KB). All other methodological details are in §6.4.

Fig. 6.9 shows the speedup provided by Confluence and Shotgun over FDIP across all nine applications. Confluence and Shotgun offer only a fraction of an ideal BTB's speedup as they are unable to cover a significant portion of all BTB misses.

We investigate the performance of these prior BTB prefetching techniques to understand why they fail to cover so many BTB misses. Since both Confluence and Shotgun leverage temporal stream prefetching to avoid BTB misses, we categorize all BTB misses into three types of temporal streams [367]: *non-repetitive*, *new*, and *recurring* streams. Temporal stream prefetching can

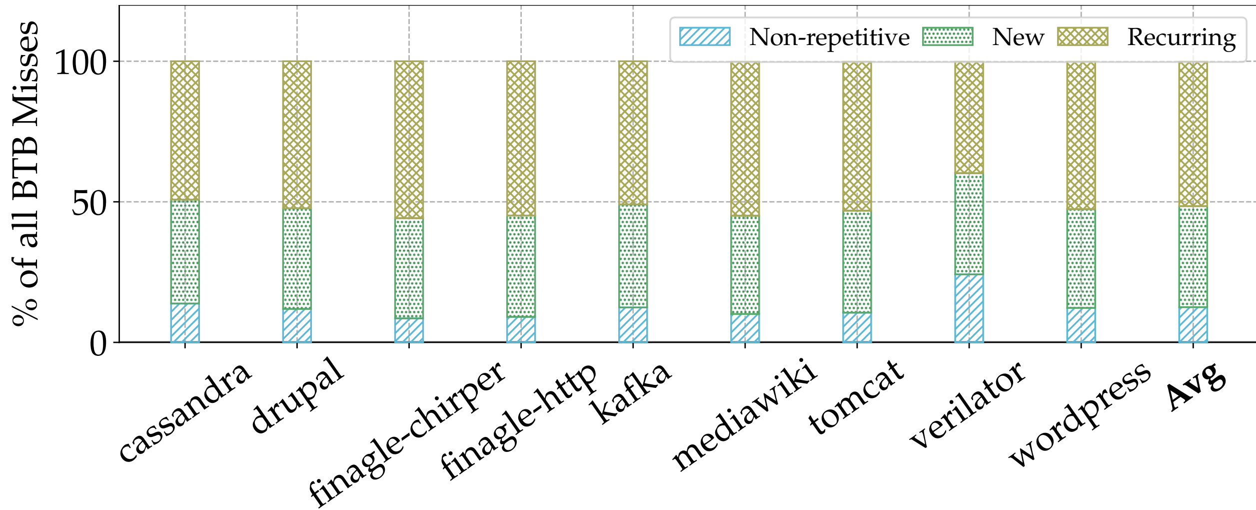


Figure 6.10: Fraction of BTB misses in temporal streams [367]

inherently cover only recurring miss streams. As shown in Fig. 6.10, while recurring miss streams constitute the majority of all BTB misses (on average 52%), new and non-repetitive streams still include a large fraction of the remaining BTB misses (on average 36% and 12% respectively) that Confluence and Shotgun do not cover. Recording access patterns at the granularity of I-cache lines instead of at the granularity of branch instructions helps Shotgun cover more BTB misses than Confluence, as Shotgun predecodes all branch instructions corresponding to a single I-cache line. Still, Shotgun falls significantly short of the ideal BTB, which we explain next.

Shotgun requires the unconditional branch footprint of the application to be small enough to fit into the BTB partition reserved for unconditional branches. Unfortunately, different applications have different unconditional branch working set sizes as we portray in Fig. 6.11. As a result, Shotgun's BTB partition for unconditional branches is too large for some applications and too small for others. Moreover, irrespective of whether an unconditional branch correlates with conditional branches, Shotgun reserves precious BTB storage bits as prefetch metadata for unconditional branches. Consequently, Shotgun wastes critical on-chip storage for some applications (*e.g.*, `drupal`, `mediawiki`, and `wordpress`) where the number of unconditional branches are much smaller than Shotgun's unconditional BTB partition size.

Shotgun incurs additional BTB misses due to one of its design constraints: the spatial range of conditional branches. Shotgun prefetches conditional branch entries based on the execution of unconditional branches. While doing so, Shotgun can only prefetch conditional branches that are within a spatial range of up to 8 cache lines of the last executed unconditional branch target. In other words, if a conditional branch resides outside this 8 cache line range, Shotgun will not be able to prefetch the corresponding BTB entry. However, as we show in Fig. 6.12, a significant portion

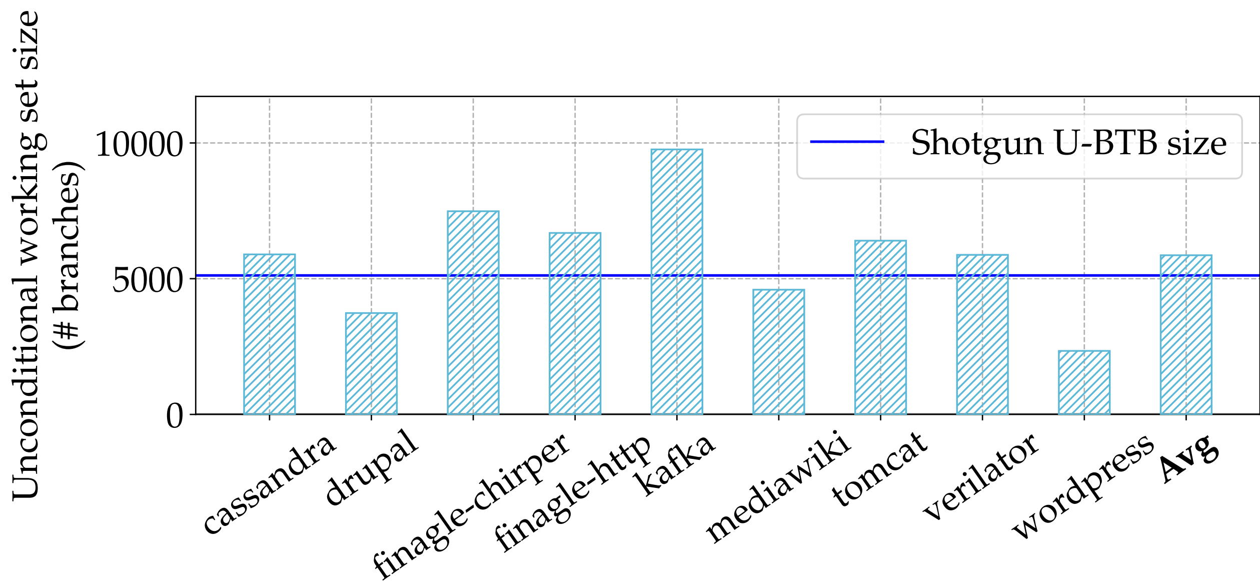


Figure 6.11: Working set size of unconditional branches and calls. Shotgun's U-BTB of 5120 entries is shown in blue.

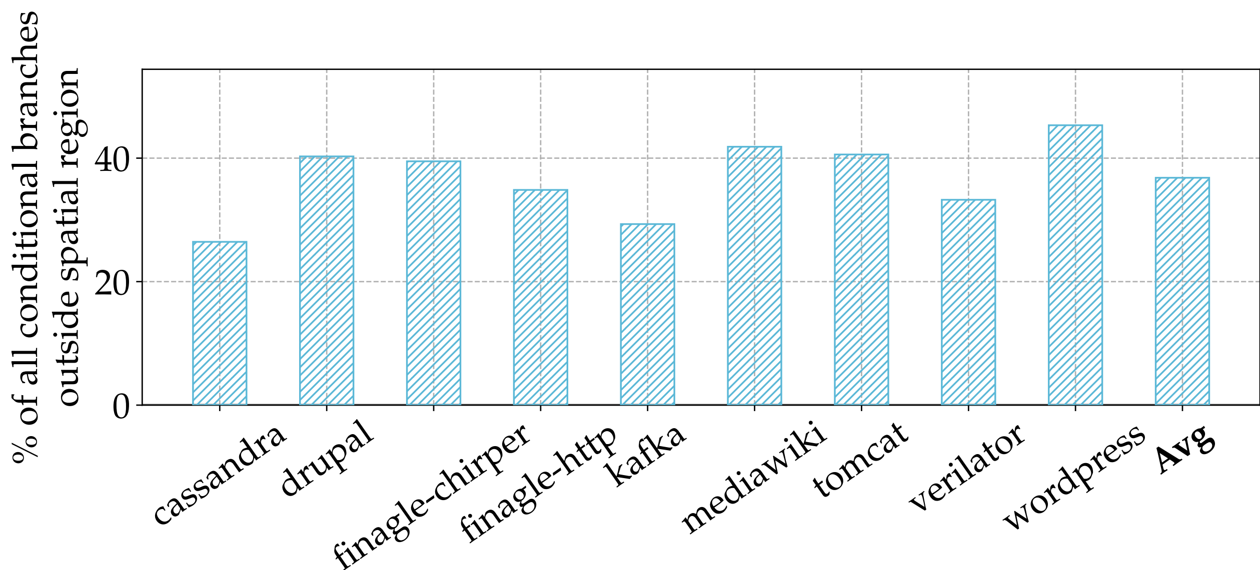


Figure 6.12: Percentage of all conditional branches that are outside the range (8 cache lines) of the last executed unconditional branch target. Shotgun cannot prefetch BTB entries for these conditional branches.

(26-45%) of all conditional branches falls outside this spatial range. Hence, Shotgun cannot cover a large portion of all BTB misses.

Based on our characterization's insights, we next present Twig, a profile-guided solution to avoid costly BTB misses.

6.3 TWIG

Modern data center application binaries are large and contain numerous unique branch instructions. These applications suffer from frequent BTB misses. Prior work addresses this issue with BTB prefetchers that require significant hardware modification and yet fail to cover a large fraction of BTB misses. We propose Twig, a profile-guided solution to prefetch BTB entries. Specifically, Twig introduces two novel techniques to avoid BTB misses. First, Twig uses a novel profile-guided mechanism to prefetch BTB entries. Second, Twig coalesces prefetch operations of multiple BTB entries into a single instruction to reduce the code bloat.

6.3.1 Software BTB Prefetching

Determining branch Program Counter (PC) and target for populating the BTB requires the processor to decode (potentially variable-length) instructions. Hardware-based BTB prefetchers such as Shotgun [211] hence need to prefetch the instructions and decode them before filling the BTB, introducing significant hardware overheads for implementing the additional pre-decoders. Additionally, the prefetch latency deteriorates if the instruction being prefetched into the BTB is not present in the processor's I-cache. Twig addresses both of these challenges. First, Twig identifies the PC and target of every direct branch instruction for an application by examining its binary. Then, Twig leverages the program's dynamic execution profile to find the branch PCs causing a large number of BTB misses. Finally, Twig modifies the application binary to prefetch corresponding BTB entries in a timely manner.

To realize Twig, we introduce a new instruction, `brprefetch` to prefetch BTB entries. The `brprefetch` instruction uses two parameters—the branch PC and the target, to insert the corresponding branch entry into the BTB. Both these fields represent instruction pointers and can be as large as 48-bit signed integers [385]. Moreover, Twig must schedule the `brprefetch` instruction early enough so that it updates the BTB before the corresponding branch target lookup occurs. We now explain how Twig meets these requirements by finding the appropriate program location to insert the `brprefetch` instruction and by storing only the address difference between the branch instruction and the target.

Prefetch injection location. Twig must insert the `brprefetch` instruction in a timely man-

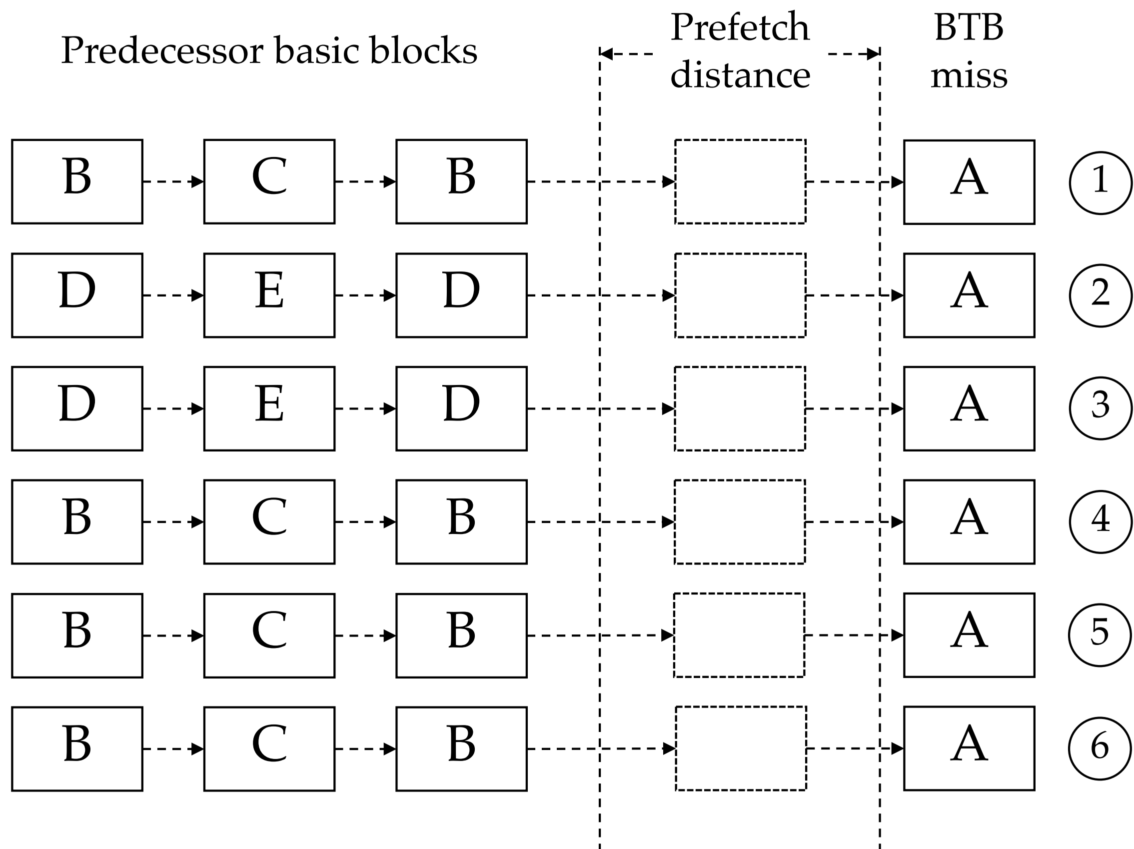
ner, ***i.e.***, the `brprefetch` instruction must retire before the corresponding branch is looked up in the BTB to avoid a BTB miss. Hence, it is critical to precisely identify the appropriate program location for inserting the `brprefetch` instruction. Twig must also emit accurate `brprefetch` instructions to avoid polluting the BTB with unnecessary entries. Since many different program paths can lead to a particular BTB miss, Twig must find the right program location to satisfy the accuracy constraint.

Twig leverages execution information to identify the appropriate program path that satisfies both the timeliness and accuracy constraint. With the help of Intel Last Branch Record (LBR) feature [7], Twig collects program execution profiles that lead to BTB misses. Intel LBR records a history of the last 32 basic blocks executed before a BTB miss along with their execution latency in cycles.

Fig. 6.13a portrays an example of such a profile for BTB misses at the branch instruction address, A, showing how Twig leverages this profile to find the injection site for the `brprefetch` instruction. This example includes six different BTB misses for A. To satisfy the timeliness constraint, Twig considers basic blocks that precede the BTB miss by at least several cycles as candidate injection sites. We call this particular cycle count the *prefetch distance*, which is one of Twig's design parameters. We use 20 cycles as the prefetch distance and evaluate Twig's sensitivity to this parameter in §6.4 (Fig. 6.26). Twig only considers predecessor basic blocks before the prefetch distance as the prefetch injection candidates. As shown in Fig. 6.13a, predecessor basic blocks B and C are considered for the BTB miss ① as they precede the BTB miss by the prefetch distance.

To satisfy the accuracy constraint, Twig computes the conditional probability of a BTB miss at A, given the execution of each candidate basic block. We show an example of this computation in Fig. 6.13b. First, Twig calculates the execution count/frequency of each candidate block using the execution profile (including BTB misses at other branch instructions apart from A). Next, Twig counts how many BTB misses at A can be avoided by inserting a prefetch instruction at the candidate injection site. Then, Twig computes the ratio of these two counts as the conditional probability of a BTB miss at A, given the execution of each candidate basic block. Finally, Twig picks the candidate with the highest conditional probability for each BTB miss as the prefetch injection site. In case of this example, Twig selects C to cover BTB misses ①, ④, ⑤, and ⑥, while Twig chooses E to avoid BTB misses ② and ③.

Prefetch target compression. The storage cost of large instruction pointers (branch PC and target) is a significant challenge for software BTB prefetching. Twig reduces this storage overhead by storing the *prefetch-to-branch-offset* instead of the entire absolute address. We define the prefetch-to-branch-offset as the delta between the prefetch instruction PC and the prefetched branch PC. Fig. 6.14 shows the quantitative insight behind this optimization. On the X-axis, we



(a) An example of profile samples for BTB misses at branch instruction address, A, containing basic block executions that precede the miss.

Basic block	Total executed	# of unique BTB misses at A that can be timely covered by the basic block	P(BTB miss at A | Basic block)
B	16	4	0.25
C	8	4	0.5
D	6	2	0.33
E	3	2	0.66

(b) An example of the conditional probability calculation to predict the BTB miss at A, given the execution of a particular basic block.

Figure 6.13: An example of how Twig analyzes BTB miss profiles to find accurate and timely prefetch injection site

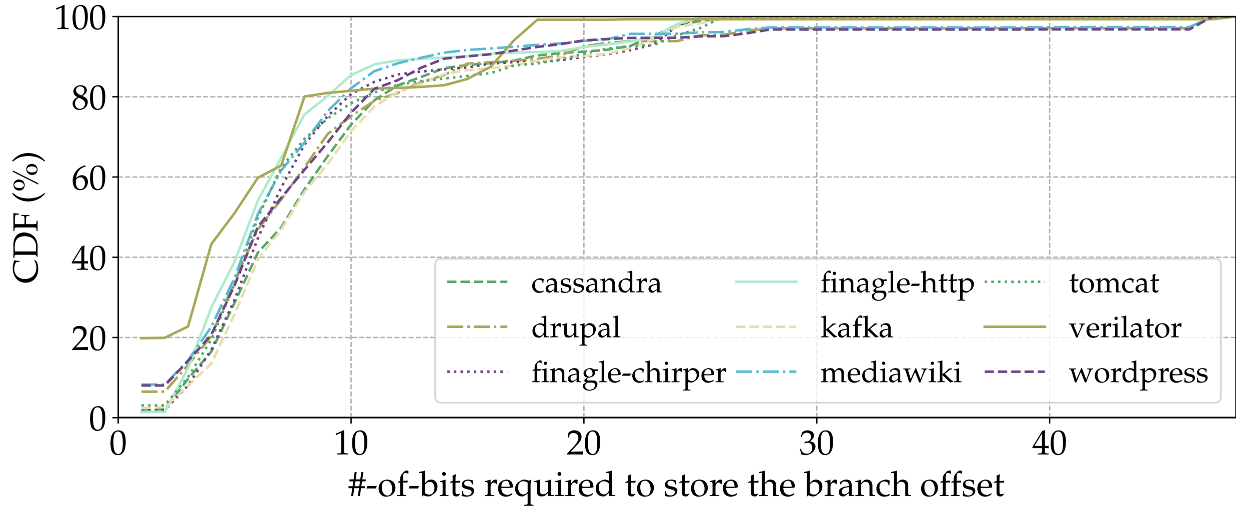


Figure 6.14: CDF of branch offset (from the prefetch injection site to the branch instruction) with variation in the number of bits required to store the offset: with just 12-bits Twig stores 80% of all branch offsets for all applications.

show the number of bits required to encode the prefetch-to-branch-offset, while on the Y-axis, we show the Cumulative Distribution Function (CDF) for all BTB misses. We find that Twig covers more than 80% of all BTB misses using just 12-bits to encode the prefetch-to-branch-offset. Twig uses the same technique to also compress the branch target. Fig. 6.15 plots the *branch-to-target-offset* on the X-axis and the CDF of all BTB misses on the Y-axis. We note that Twig again covers 80% of all BTB misses for most applications using just 12-bits. Only for `verilator`, covering more than 80% of all BTB misses requires larger than 12-bit signed integers. To cover the remaining BTB misses and to optimize the storage overhead even further, Twig proposes BTB prefetch coalescing that we describe next.

6.3.2 BTB Prefetch Coalescing

Branch instructions with large address differences cannot directly be encoded using the prefetch instruction introduced in §6.3.1. For these too-large-to-encode branch instructions, Twig stores the addresses of the branch instruction and the target as key-value pairs in memory. Twig stores these pairs in sorted order based on the branch instruction address. Storing branch entries in sorted order helps Twig leverage spatial locality among different entries. The key-value pairs are generated at compile time and added to the instruction binary as part of the text segment.

Twig introduces the `brcoalesce` instruction that takes the address of a key-value pair as a parameter and prefetches the corresponding entry to the BTB. To improve its efficiency, `brcoalesce` includes an n-bit bitmask as an additional parameter to prefetch multiple consecutive

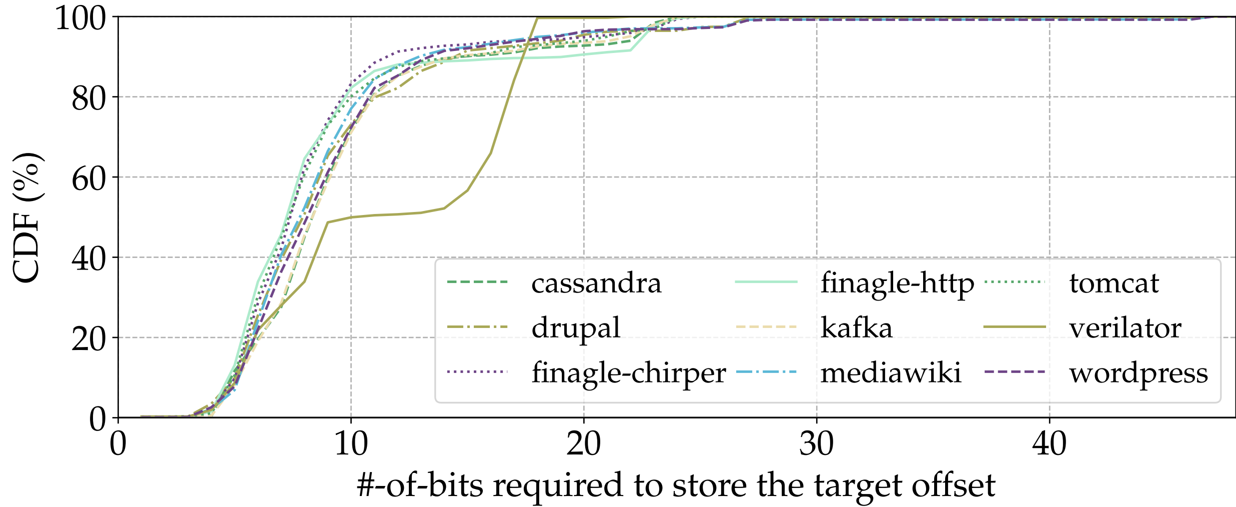


Figure 6.15: CDF of branch target offset with variation in the number of bits required to store the offset: with just 12-bits Twig stores 80% of all branch targets for most applications.

entries (for this reason, the key-value pairs are sorted in memory). Coalescing enables prefetching of multiple too-large-to-encode BTB entries with a minimal increase in the instruction footprint.

The size of the bitmask, n, is another design parameter. With a smaller bitmask, Twig would be able to prefetch only a small number of correlated BTB entries. With a larger bitmask, Twig can coalesce more prefetch operations. We investigate the impact of the bitmask size on the effectiveness of BTB prefetch coalescing in §6.4 and show that Twig achieves a majority of the performance benefit with just an 8-bit bitmask (Fig. 6.27).

6.4 Evaluation

In this section, we first describe (1) our experimental setup to collect execution profiles for our target data center applications, (2) different application input configurations, and (3) our simulation infrastructure. Then, we evaluate Twig using several key performance metrics.

6.4.1 Methodology

Data center applications and inputs. We evaluate Twig in the context of nine popular data center applications (as described in §6.2). We evaluate these applications with different input configurations such as the input data size, the webpage requested by the client, the number of client requests per second, random number seeds, and the number of server threads. Since Twig's profile-guided optimizations depend on the application input, we optimize each of these applications using the profile from one input and test the performance of the optimization on a different input.

Table 6.1: Simulator Parameters

Parameter	Value
CPU	3.2GHz, 6-wide OOO, 24-entry FTQ, 224-entry ROB, 97-entry RS
Branch prediction unit	64KB TAGE-SC-L [317] (up to 12-instruction), 8192-entry 4-way BTB, 32-entry RAS, 4096-entry 4-way IBTB
Memory hierarchy	32KB 8-way L1i, 32KB 8-way L1d, 1MB 16-way unified L2, 10MB 20-way shared L3 per socket

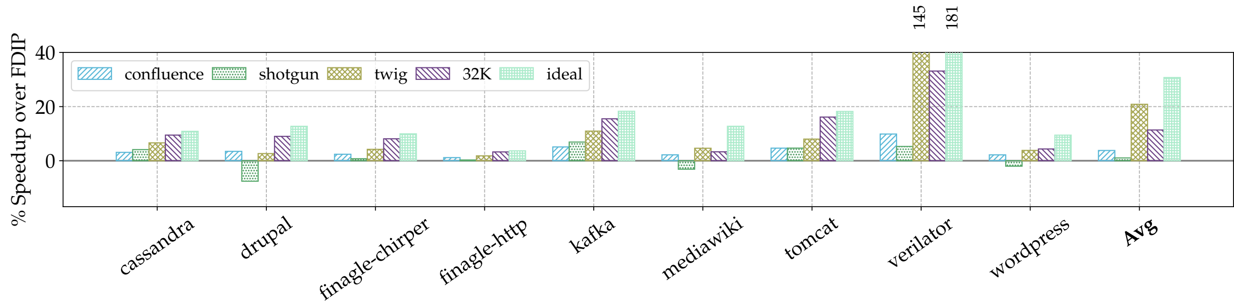


Figure 6.16: Percentage speedup over the FDIP baseline: 32K is for a 32K-entry BTB compared to the 8K-entry baseline BTB. Twig outperforms even the 32K-entry BTB on average with just an 8K-entry BTB with prefetching.

Profile collection. We leverage Intel's "`baclears.any`" hardware performance event along with LBR [7] to collect the application execution context profiles that lead to a BTB miss.

Simulation and trace collection. We evaluate Twig using Scarab [14]. In Scarab, we implement support for the BTB prefetch instructions (`brprefetch` and `brcoalesce`) and also add implementations for FDIP, Shotgun, and Confluence. We list different simulation parameters that resemble a recent state-of-the-art industry baseline [150, 151] in Table 6.1. Both trace-driven and execution-driven Scarab modes use Intel PIN [237], which cannot instrument kernel mode instructions. To support kernel mode instruction simulations, we collect application traces using Intel Processor Trace [1] and modify Scarab to support simulating such traces as well. We simulate traces of 100 million representative, steady-state instructions for each data center application.

6.4.2 Performance Analysis

We now validate Twig's effectiveness using key performance metrics. First, we compare Twig's speedup to the speedup offered by an ideal BTB and the state-of-the-art BTB prefetcher, Shotgun [211]. Then, we evaluate the individual speedup contributions of software BTB prefetching

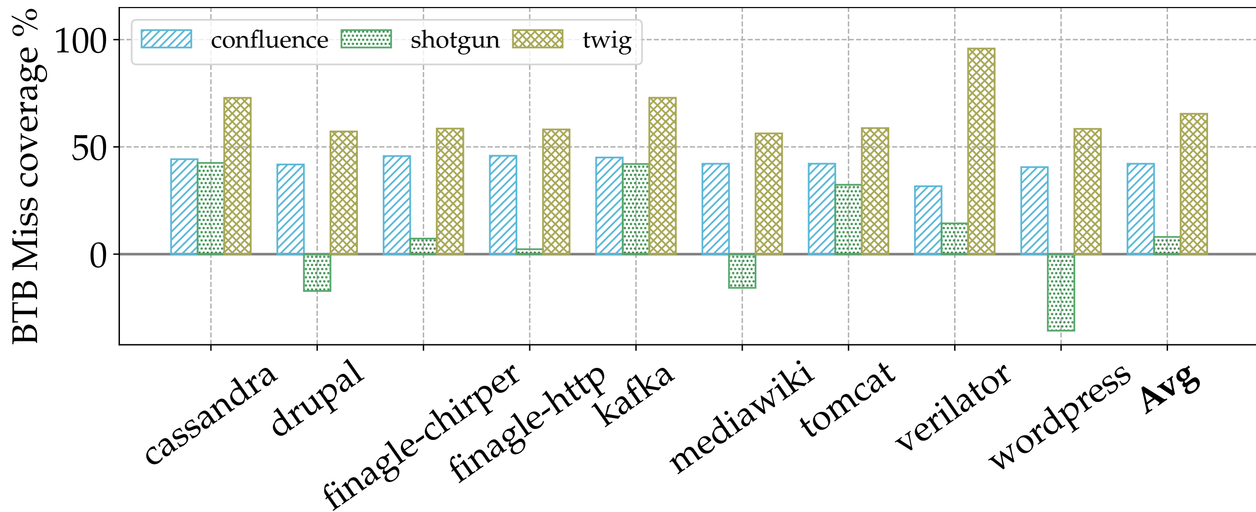


Figure 6.17: BTB miss coverage of Twig, Confluence, and Shotgun: on average Twig covers 65.4% of all BTB misses.

and BTB prefetch coalescing. We also compare Twig against Shotgun in terms of BTB miss coverage and BTB prefetch accuracy. Furthermore, we compare speedups achieved by Twig and Shotgun across different application inputs. Finally, we measure Twig's static and dynamic overhead due to the additional BTB prefetch instructions.

Speedup. We show Twig's speedup (brown bars) for nine data center applications in Fig. 6.16. For comparison, we also show speedup offered by an ideal BTB (purple bars) and state-of-the-art BTB prefetcher, Shotgun (green bars). As shown, Twig achieves on average 20.86% speedup compared to 31% mean speedup achieved by an ideal BTB and 1% mean speedup achieved by Shotgun. On average, Twig achieves 48% (and up to 80%) of the speedup achieved by an ideal BTB that incurs no BTB misses. Twig cannot provide the entire benefit (100%) of an ideal BTB for a number of reasons. First, some BTB misses do not have a predecessor basic block that can predict the potential BTB miss with high accuracy. Second, BTB prefetch instructions injected by Twig incur both static and dynamic instruction overheads (we quantify this overhead later in this section). Finally, Twig cannot cover some previously unobserved BTB misses due to the use of different inputs in profiling and testing (we also quantify this later in the section). Still, Twig advances the state-of-the-art by outperforming Shotgun by 19.82% on average (and up to 139.8%) as Twig covers more BTB misses than Shotgun.

BTB miss coverage. Fig. 6.17 shows the BTB miss coverage comparison between Twig and Shotgun. As shown, Twig covers on average 65.4% (and up to 95.84%) of all BTB misses. Additionally, Twig covers on average 57.4% (and up to 94%) more BTB misses than the state-of-the-art prefetcher, Shotgun. Twig outperforms Shotgun to cover 57.4% more BTB misses primarily because of the reasons we describe in §6.2.3. In contrast to Shotgun's ability to prefetch only condi-

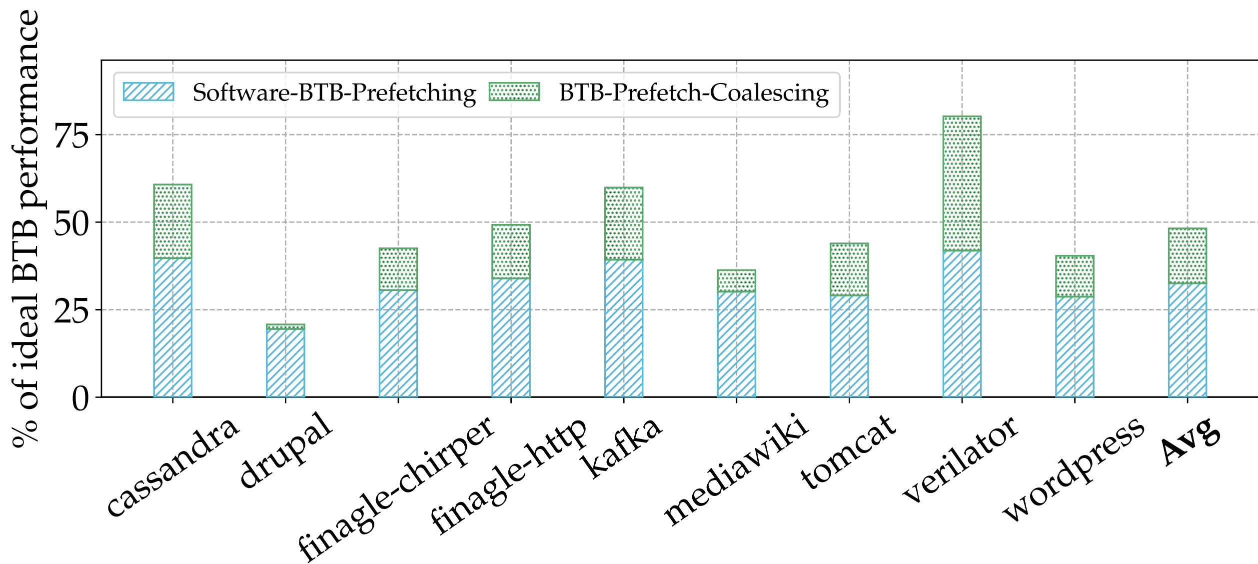


Figure 6.18: Contribution of software BTB prefetching and BTB prefetch coalescing toward Twig performance of an ideal BTB: software BTB prefetching provides greater benefits than BTB prefetch coalescing across applications.

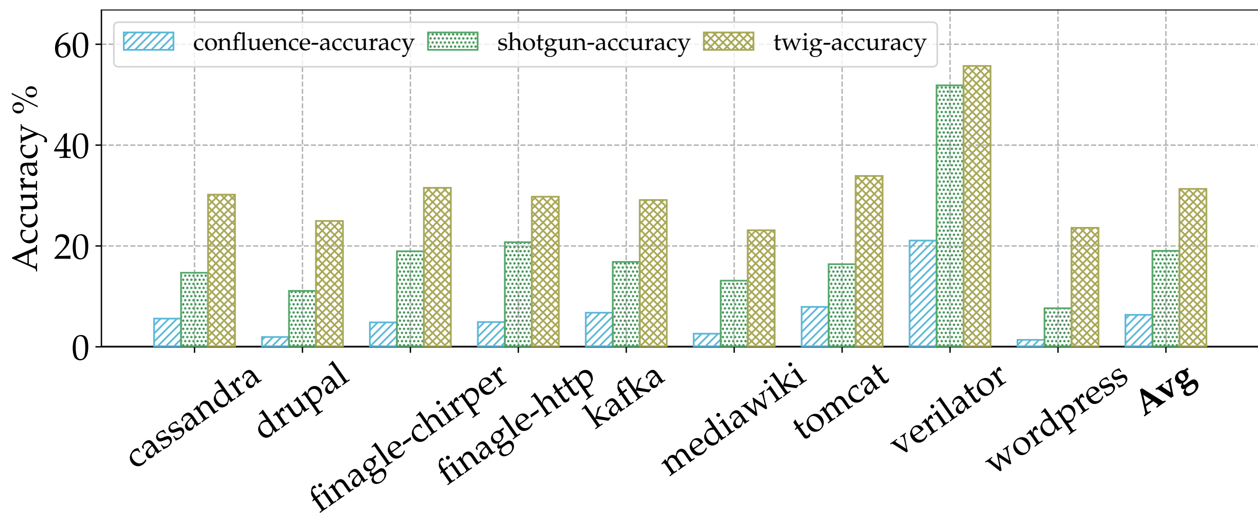


Figure 6.19: Prefetch accuracy of Twig, Confluence, and Shotgun: on average Twig provides 31.3% BTB prefetch accuracy across nine data center applications.

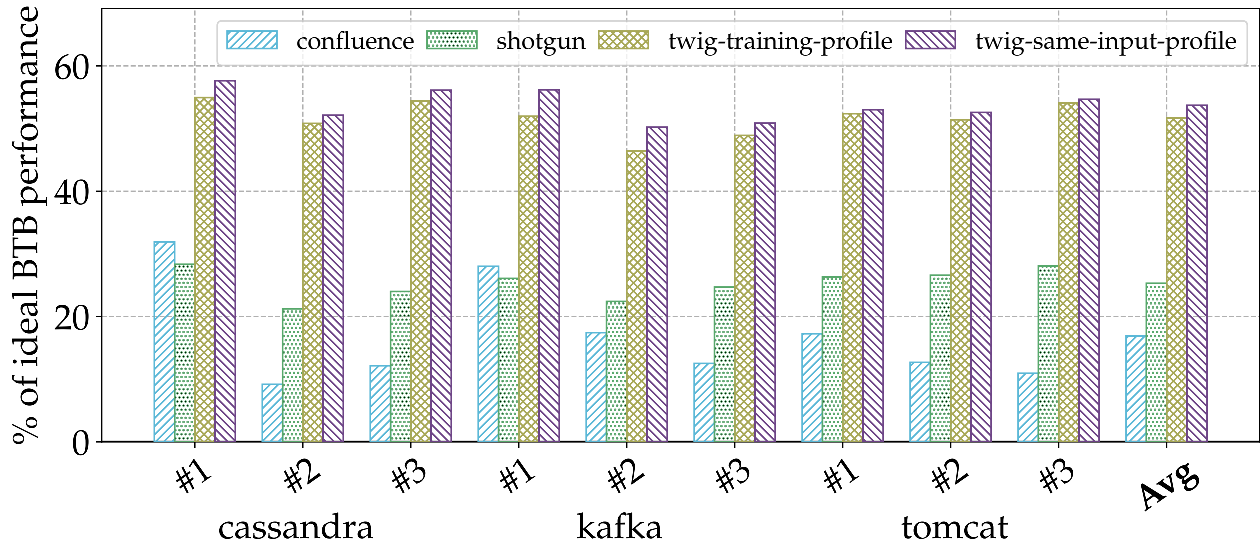


Figure 6.20: Twig's speedup across different application inputs as the percentage of an ideal BTB performance: Twig trained on a different input provides performance benefits comparable to Twig trained on the same input and outperforms existing BTB prefetching mechanisms.

tional branch entries within a limited spatial range, Twig can prefetch BTB entries irrespective of branch type or distance.

Performance of software BTB prefetching and BTB prefetch coalescing. Fig. 6.18 shows the individual contributions of software BTB prefetching and BTB prefetch coalescing to Twig's overall speedup. As shown, software BTB prefetching without any coalescing provides on average 32.6% speedup (70.9% of overall performance gains) across different applications. On top of this, prefetch coalescing provides on average 15.7% speedup (29.1% of overall benefits) by reducing the static and dynamic instruction overhead.

Prefetch accuracy. We show Twig's prefetch accuracy in Fig. 6.19 and compare it against Shotgun's prefetch accuracy. As shown, Twig provides 31.3% average accuracy. Moreover, Twig achieves 12.3% higher prefetch accuracy than Shotgun due to the fundamental limitation of hardware temporal stream prefetching. Like most prior hardware techniques on temporal memory streaming [369, 367, 114, 368, 332, 60], Shotgun remembers the spatial footprint seen during the last execution and prefetches the corresponding BTB entries. While prefetching the most recently executed footprint is efficient in terms of metadata storage (compared to most frequently executed footprint), it incurs many inaccurate BTB prefetches. Twig, on the other hand, leverages a large amount of execution information from the collected profile to identify the most accurate prefetch predecessor and achieves higher prefetch accuracy.

Performance across different application inputs. The effectiveness of profile-guided optimizations usually depends on the corresponding application input. To investigate how this depen-

Table 6.2: Twig's average speedup across different application inputs with standard deviations.

Application	% of ideal BTB performance			
	Same input profile		Training profile	
	Average	Standard deviation	Average	Standard deviation
cassandra	49.31	10.04	45.93	15.53
drupal	36.77	14.31	43.15	9.84
finagle-chirper	38.30	9.13	31.99	10.29
finagle-http	34.03	7.73	32.66	5.62
kafka	52.35	2.17	49.93	2.26
mediawiki	38.78	10.95	43.78	5.11
tomcat	51.25	4.02	45.77	15.84
verilator	80.33	0.39	79.19	0.33
wordpress	45.15	14.69	49.71	12.85

dence affects Twig's performance, we compare the speedups achieved by Twig across different application inputs in Fig. 6.20. For each application, we use the profile from input `'#0'` to optimize BTB performance using Twig and measure the speedups for other inputs, `'#1, #2, #3'`. For comparison, we also measure speedups achieved by Twig when optimized with the profile from the same input. Finally, we compare Twig against Confluence and Shotgun for different application inputs. For each configuration, we normalize the overall speedup by expressing it in terms of ideal BTB performance.

As shown in Fig. 6.20, Twig provides significantly more benefit than state-of-the-art mechanisms [190, 211] even while using profiles from a different application input. Twig provides a greater speedup when optimized using input-specific profiles (as shown in Table 6.2) for 6 out of 9 applications. However, for the remaining three applications, Twig can achieve even better speedup with profiles from a different application input. Nonetheless, Twig achieves comparable speedups with profiles from both same and different inputs.

Prefetch overhead. Twig does not introduce any extra metadata storage. Therefore, instructions added to perform BTB prefetching are the only overhead Twig introduces. We quantify the static and dynamic overhead of these prefetch instructions in Fig. 6.21 and 6.22. In Table 6.3, we quantify the combined overhead of static and dynamic instruction increase based on working set size increase in terms of the number of added bytes. As shown, Twig introduces less than 8% static and 12.6% dynamic instruction overhead for all cases. Specifically, Twig incurs the highest dynamic overhead for `verilator` to cover the large number of BTB misses incurred by the application (BTB MPKI of 121).

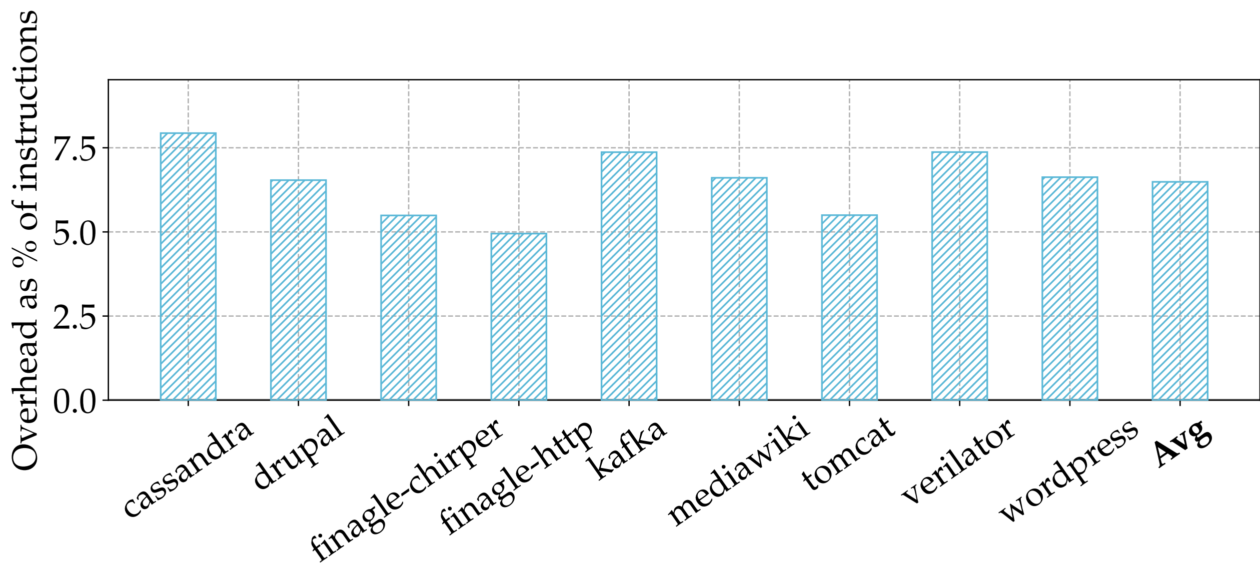


Figure 6.21: Static overhead of Twig, measured in % of additional instructions in the binary for a given workload: on average Twig inserts 6% extra static instructions.

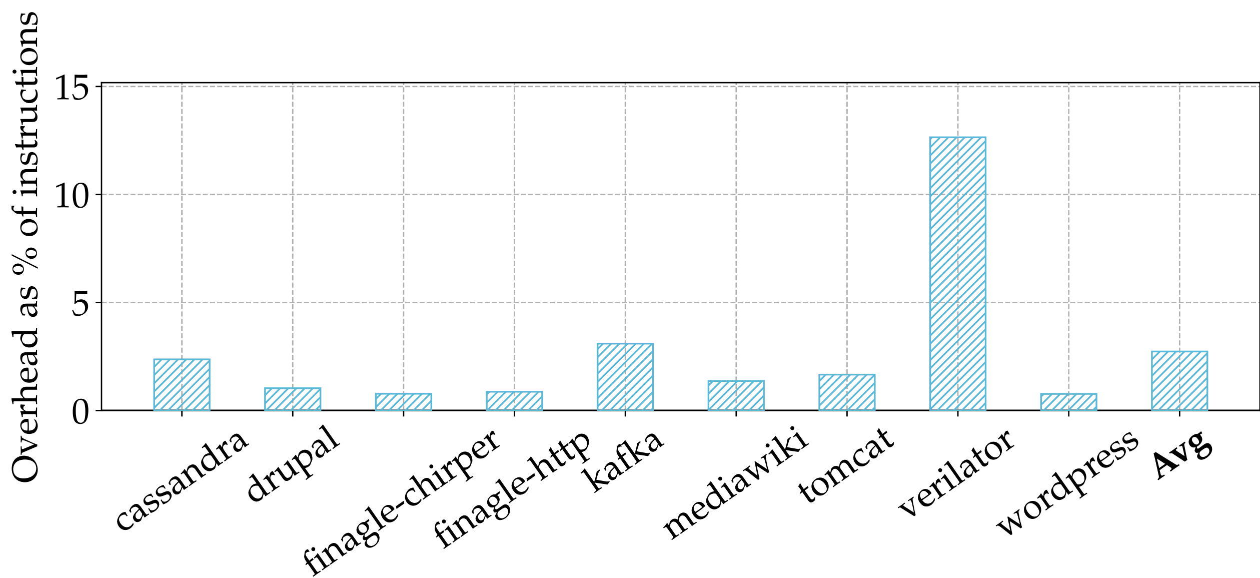


Figure 6.22: Dynamic overhead of Twig, measured in % of additional executed instructions for a given workload: on average Twig incurs only 3% extra dynamic instructions.

Table 6.3: Instruction working set size overhead of Twig.

Application	Instruction working set size (MB)	Additional instruction size (MB)	Overhead (%)
cassandra	4.23	0.26	6.08
drupal	1.75	0.05	2.93
finagle-chirper	2.05	0.07	3.54
finagle-http	5.29	0.42	7.97
kafka	3.28	0.16	4.78
mediawiki	2.24	0.08	3.70
tomcat	2.40	0.10	4.10
verilator	13.56	1.34	9.86
wordpress	1.93	0.06	3.09

6.4.3 Sensitivity Analysis

We investigate the sensitivity of different design parameters on Twig's effectiveness. First, we compare the speedup achieved by Twig and Shotgun for different BTB storage budgets (size and associativity) and prefetch buffer sizes. Additionally, we evaluate the effect of changing the prefetch distance and FDIP run-ahead on Twig's effectiveness.

BTB storage budget. In Fig. 6.23, we evaluate how sensitive Twig is to the storage budget allocated to the BTB by varying the number of BTB entries. We fix all other parameters and vary the number of BTB entries between 2048 (2K) and 65536 (64K). As Fig. 6.23 shows, Twig achieves more speedup than either Shotgun or Confluence across all BTB sizes. We also vary BTB's associativity from 4 ways per set to 128 ways per set. Fig. 6.24 shows how Twig outperforms both Shotgun and Confluence for any associativity.

Prefetch buffer size. We next vary the size of the BTB prefetch buffer. This enables us to hold additional BTB entry candidates at any given time, enabling Twig prefetches to not evict each other. As shown in Fig. 6.25, Twig's performance scales from from 8 to about 128 entries before it begins to experience diminishing returns. Shotgun and Confluence do not experience this same scaling, indicating that Twig provides greater benefits than prior works irrespective of the prefetch buffer size.

Prefetch distance. Fig. 6.26 shows how Twig's effectiveness varies in response to variation in prefetch distance. We vary the prefetch distance from 0 to 50 cycles and measure Twig's average performance as a percentage of ideal BTB performance across applications. As shown, Twig provides only a portion of the potential speedup when the prefetch distance is too small to complete the prefetch before the BTB lookup. On the other hand, Twig cannot find an appropriate prefetch

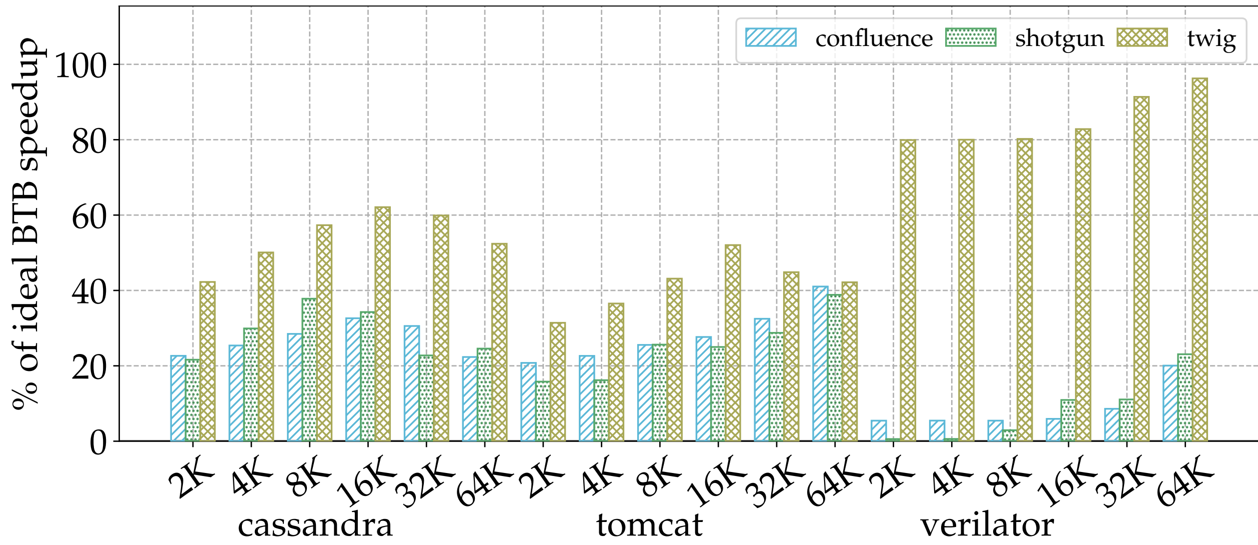


Figure 6.23: % of speedup obtained by Twig compared to an ideal BTB for BTB capacities ranging from 2048 entries to 65536 entries

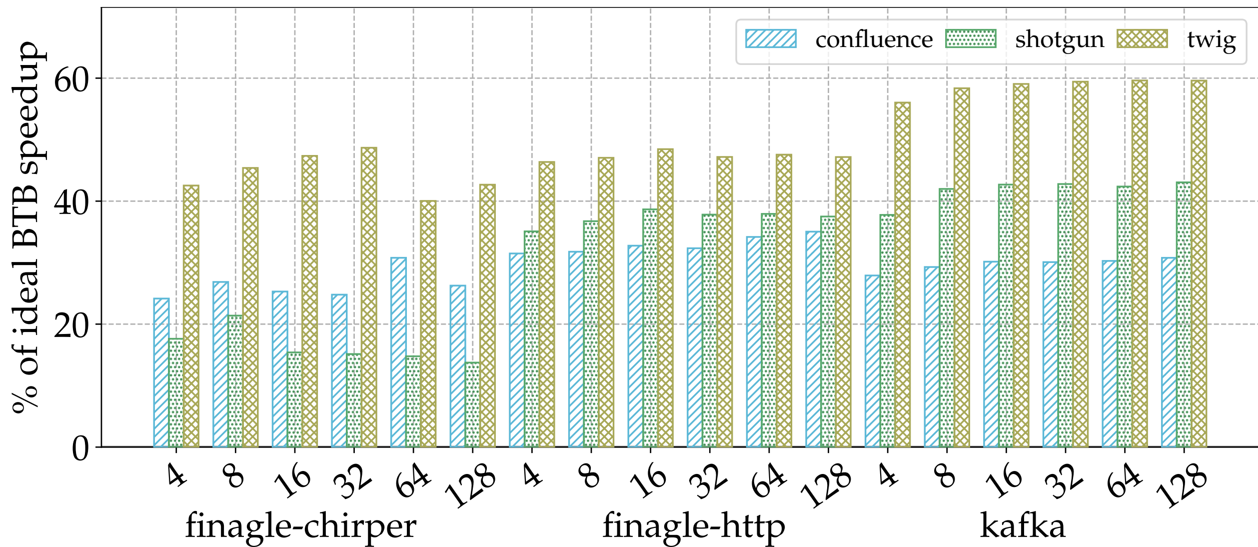


Figure 6.24: % of speedup obtained by Twig compared to an ideal BTB for BTB associativity ranging from 4 to 128

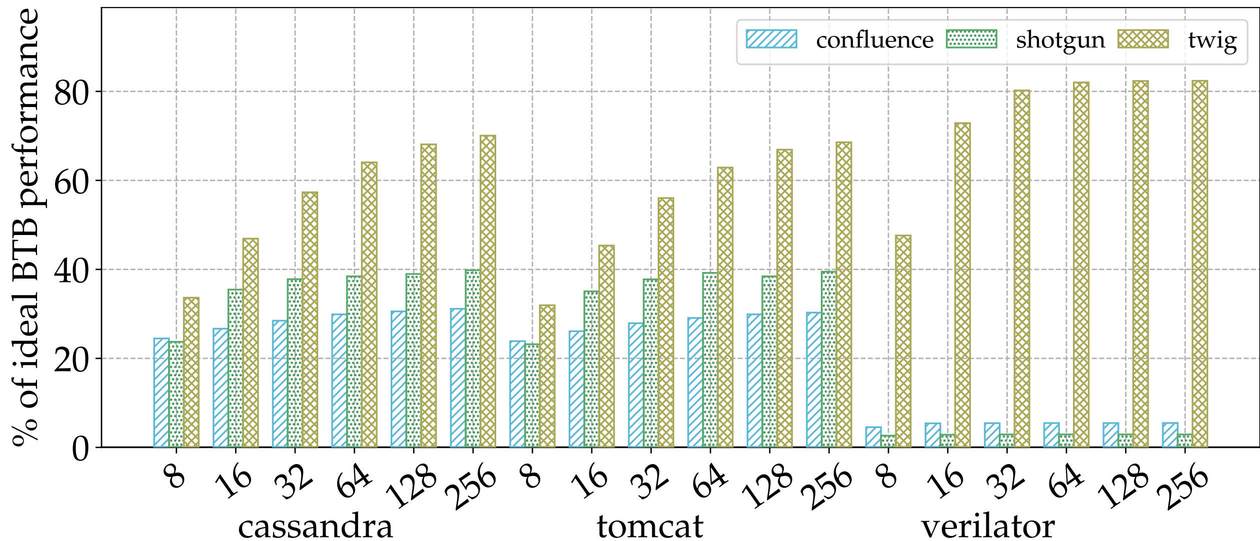


Figure 6.25: Percent of speedup obtained by Twig compared to an ideal BTB for the size of the prefetch buffer, ranging from 8 to 256

injection site when the prefetch distance is too large to ignore accurate predecessors. Consequently, Twig provides the greatest benefit with 15-25 cycles of prefetch distances across different applications.

Coalescing size. We investigate the effectiveness of Twig's BTB prefetch coalescing with an increase in coalescing bitmask size. Fig. 6.27 shows the average performance gains of BTB prefetch coalescing as the percentage of ideal BTB performance for different bitmask sizes (1-bit to 64-bit) across nine data center applications. As shown, Twig realizes a large fraction of the potential speedup with an 8-bit bitmask. Consequently, we use 8-bits to coalesce BTB prefetch instructions.

FDIP Run-ahead. Finally, we vary the size of the Fetch Target Queue (FTQ), which determines how far ahead the decoupled frontend can run. We vary the FTQ size from 1 to 64 branches. Fig. 6.28 shows that Twig achieves a similar performance relative to ideal at every measured FTQ length. Since a longer FTQ has been shown to improve performance by reducing frontend stalls [150], this result implies that Twig scales well to frontends that run far ahead of the fetch unit.

6.5 Related Work

Preventing I-cache misses. Many prior works focus on reducing frontend stalls via eliminating I-cache misses. These techniques can be summarized in three distinct categories: software only, hardware only, and a hybrid software/hardware approach. Software techniques include improving instruction locality via basic block/function reordering [289, 276], hot/cold splitting [78], and other

Figure 6.26: Twig's average performance variation in response to increasing the prefetch distance. Across different applications, Twig provides greatest benefit with prefetch distance 15-25 cycles.

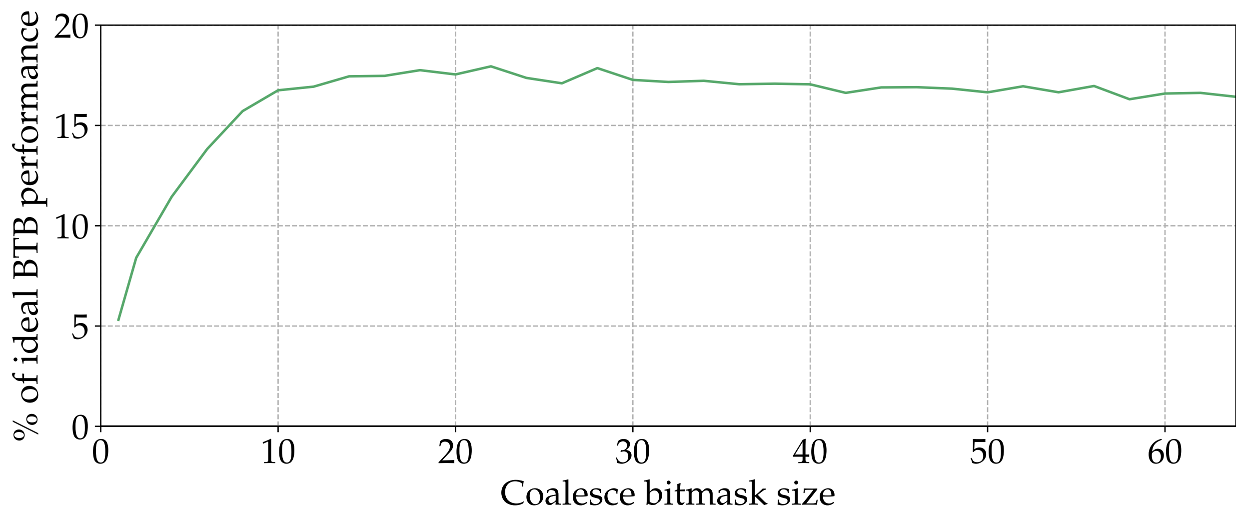


Figure 6.27: Twig's average performance variation in response to changes in the coalesce bitmask size. Twig achieves a majority of the potential performance gains with a 8-bit coalesce bitmask.

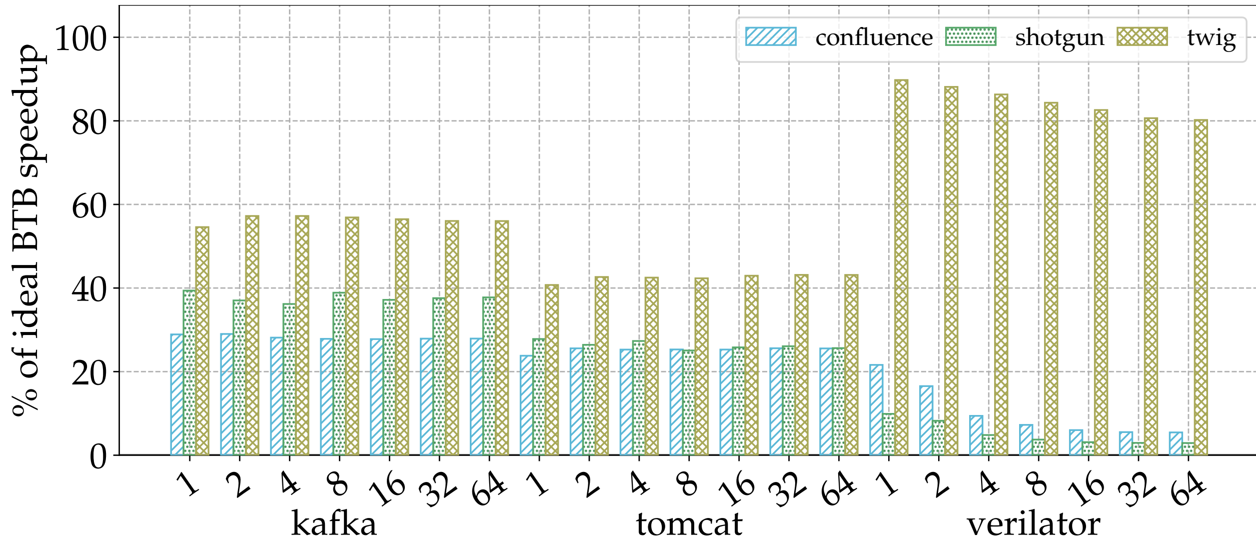


Figure 6.28: % of speedup obtained by Twig compared to an ideal BTB for the size of the FTQ, or maximum distance the decoupled frontend can run ahead, varied between 1 and 64

Profile-Guided Optimizations (PGO) [65, 238, 136, 299, 410, 286, 40, 137, 57, 221, 236, 278]. Improved layout techniques are only able to eliminate a subset of all I-cache misses and finding the optimal code layout for I-cache performance is intractable in practice [287, 41]. Hardware-only techniques tend to have one of two limitations. Either the techniques have prohibitive on-chip storage costs [113, 114], or they end up being significantly more complex [189, 190, 209] than prefetching techniques implemented in real hardware [306, 302]. Hybrid hardware and software approaches [41, 199] attempt to avoid the pitfalls of software only or hardware only approaches by performing the complicated software analysis ahead of time and executing simple prefetch instructions at runtime. However, prior approaches either make assumptions that are too simplistic, limiting prefetching accuracy, or execute too many dynamic instructions which exacerbate the application's code footprint [41, 266]. State-of-the-art I-cache prefetchers include the SN4L+Dis+BTB design [34] and the contenders of the first instruction cache prefetching competition (IPC-1) [268, 134, 33, 320, 254, 305, 128, 122]. Of the above proposals, FDIP has a desirable trade-off between metadata cost and prefetching effectiveness [212, 211]. Even with significantly smaller metadata storage costs, FDIP provides comparable performance benefits to state-of-the-art I-cache prefetchers [150, 151]. Moreover, recent commercial CPU designs adopted some FDIP variants to reduce frontend stalls [350, 283, 308, 129]. Therefore, in this work, we focus on improving FDIP effectiveness by introducing software BTB prefetching that provides 20.86% average speedup without requiring any extra metadata storage.

BTB redesign / compression. The design and usage of the storage allocated to the BTB has long been debated. BTB entries commonly hold some combination of a tag, prediction informa-

tion, and target address [218, 284]. The basic-block style BTB also contains the address of the fall-through basic block [398]. Compressing BTB entry size is common to enable the BTB to host more entries in the same storage budget. Such techniques [107, 322, 208, 301, 61, 165, 284] include using fewer bits for the tag, removing the page number from the tag, encoding the branch target as a small delta from the branch PC, and adding a larger second level BTB for which the first level BTB acts as a small cache. BTB-X [37] and PDede [337] apply several of these compression techniques, including partitioning the BTB into segments to enable aggressive compression and deduplication. All of these techniques enable the underlying BTB to have a larger capacity for a given storage budget. Since Twig prefetches entries into the BTB, it is independent of the underlying BTB and should be just as effective with the above techniques.

BTB prefetching. Phantom-BTB (PBTB) [60] virtualized predictor metadata into the shared L2 cache, and used entries in the virtualized table to prefetch BTB entries. PBTB suffers from a relatively high cost of metadata storage and a longer latency access time for important branch prediction metadata. Two-level bulk preload [53] maintains two BTB levels per-core, with a mechanism to fetch a group of BTB entries for a fixed-size region to the first level on a miss to any branch in that region. This is limited to exploiting the available spatial locality of a branch, and thus is similar to the next-line prefetchers. Confluence [190] keeps the I-cache and BTB contents in sync via their AirBTB design, with the ability to predecode branches and BTB entries for a given I-cache block. Locking the I-cache and BTB contents limits the runahead ability of the branch predictor unit. Moreover, Confluence relied on a metadata-expensive temporal prefetcher, SHIFT [190, 212, 189]. Boomerang [212] modifies FDIP to predecode fetched I-cache blocks and insert the corresponding BTB entries. However, the ability for these entries to be timely is largely dependent upon the frontend to run far enough ahead, and miss coverage suffers when there are many BTB misses [211]. Shotgun [211] partitions the BTB into the Unconditional BTB (U-BTB) and much smaller Conditional BTB (C-BTB), with a way to prefetch entries into the C-BTB when the U-BTB is hit. As such, Shotgun relies on a high U-BTB hit rate to keep the C-BTB full of useful entries [34]. This reliance limits Shotgun's ability to scale. Additionally, any fixed partitioning scheme, as in U-BTB vs. C-BTB sizes, need the workload's distribution of branches to match, and results in underutilized space when the application deviates from the fixed partitioning scheme. See §6.2.3 for a in-depth investigation on the impact of the limitations of each approach, and why they cannot cover all BTB misses. In this work, we investigate the reasons behind their limitation and address such limitations by proposing profile-guided BTB prefetch mechanisms that outperform prior techniques.

6.6 Conclusion

Large branch footprints of data center applications cause frequent BTB misses, resulting in significant frontend stalls. We showed that existing BTB prefetching techniques fail to overcome these stalls due to inadequate understanding of the applications' branch access patterns. To address this limitation, we proposed Twig, a profile-guided BTB prefetching mechanism. Twig presents two BTB prefetching techniques: software BTB prefetching and BTB prefetch coalescing. We evaluated Twig in the context of nine popular data center applications. Across these applications, Twig achieves an average of 20.86% (2%-145%) performance speedup and outperforms the state-of-the-art BTB prefetching technique by 19.82%.

CHAPTER 7

Whisper: Profile-Guided Branch Misprediction Elimination for Data Center Applications

Modern data center applications experience frequent branch mispredictions – degrading performance, increasing cost, and reducing energy efficiency in data centers. Even the state-of-the-art branch predictor, TAGE-SC-L, suffers from an average branch Mispredictions Per Kilo Instructions (branch-MPKI) of 3.0 (0.5-7.2) for these applications since their large code footprints exhaust TAGE-SC-L's intended capacity.

In this work, we[1] propose Whisper, a novel profile-guided mechanism to avoid branch mispredictions. Whisper investigates the in-production profile of data center applications to identify precise program contexts that lead to branch mispredictions. Corresponding prediction hints are then inserted into code to strategically avoid those mispredictions during program execution. Whisper presents three novel profile-guided techniques: (1) *hashed history correlation* which efficiently encodes hard-to-predict correlations in branch history using lightweight Boolean formulas, (2) *randomized formula testing* which selects a locally-optimal Boolean formula from a randomly selected subset of possible formulas to predict a branch, and (3) the extension of Read-Once Monotone Boolean Formulas with *Implication and Converse Non-Implication* to improve the branch history coverage of these formulas with minimal overhead.

We evaluate Whisper on 12 widely-used data center applications and demonstrate that Whisper enables traditional branch predictors to achieve a speedup close to that of an ideal branch predictor. Specifically, Whisper achieves an average speedup of 2.8% (0.4%-4.6%) by reducing 16.8% (1.7%-32.4%) of branch mispredictions over TAGE-SC-L and outperforms the state-of-the-art profile-guided branch prediction mechanisms by 7.9% on average.

[1]Some of the work in this chapter was performed in collaboration with Muhammed Ugur, Krishnendra Nathella, Dam Sunwoo, Heiner Litz, Daniel A Jiménez, and Baris Kasikci [200]. Therefore, I use the "we" pronoun in this chapter to acknowledge their involvement in this work.

7.1 Introduction

Modern data center applications exhibit large instruction footprints and suffer from frequent frontend and misprediction[2] stalls, incurring performance losses worth millions of dollars [279, 275, 41, 344, 39, 179, 112]. These applications contain complex application logic [275, 279, 41] and frequently use different libraries [179], language runtimes [23, 274], and kernel modules [41, 361]. As a result, these applications' hot code footprints range from tens to hundreds of megabytes [179, 41, 278, 279] which overwhelm on-chip cache structures like the Instruction cache (I-cache), Branch Target Buffer (BTB), and the branch predictor, whose sizes are only hundreds of kilobytes [39]. Consequently, processors are unable to sufficiently fetch useful instructions [41] when executing modern data center applications – leading to frequent frontend and misprediction stalls [39]. These stalls notably increase the Total Cost of Ownership (TCO) of a data center [179, 41], and even a single-digit reduction of these stalls can save millions of dollars in management and energy costs while significantly reducing the global carbon footprint [344].

Several techniques have been proposed to address these challenges including decoupled frontends [301] leveraging Fetch Directed Instruction Prefetching (FDIP) [302, 150, 151] and Profile-Guided Optimizations (PGO) [289, 65, 278, 199, 196, 202, 195] that are efficiently supported by today's hardware [350, 283, 308, 129, 396] and software [65, 41, 126, 279, 263] systems.

On the hardware side, FDIP avoids the tight coupling between branch prediction and instruction fetch, enabling branch predictor-guided instruction prefetching to avoid frontend stalls. As long as FDIP can run sufficiently ahead, it can eliminate frontend stalls effectively. Thereby, FDIP's performance depends on the accuracy of the branch predictor, as frequent mispredictions limit FDIP's effectiveness in mitigating frontend stalls [114, 109, 212, 211].

Profile-guided code layout optimizations address the large instruction footprint problem by placing frequently executed I-cache lines together, thereby improving instruction locality. These techniques do not require any hardware modifications, and although these techniques are sensitive to profile quality [139], they work well in practice. Profiles for data center applications change slowly over several weeks [65] while companies like Google and Facebook deploy new binaries every few days – giving PGO techniques ample opportunity to adapt to changing profiles [65, 278, 279]. As a result, these techniques are widely-used in today's data centers [65, 278, 41, 126, 279]. For example, half of all CPU cycles in Google data centers execute instructions from PGO-optimized applications [65]. Unfortunately, existing PGO techniques primarily reduce frontend stalls and eliminate less than 10% of all branch mispredictions [278].

To quantify the performance implications of branch mispredictions, we extensively investigate the behavior of 12 modern data center applications to show that their large code footprints trigger

[2] we use 'branch misprediction' and 'misprediction' interchangeably

frequent branch mispredictions, significantly impeding the efficacy of state-of-the-art techniques. In particular, we find that even a 64KB TAGE-SC-L [317] predictor experiences an average branch-MPKI of 3.0 (0.5-7.2) for these applications primarily due to capacity reasons. Furthermore, our investigation reveals that state-of-the-art profile-guided branch prediction mechanisms, BranchNet [405] and Read-Once Monotone Boolean Formulas (ROMBF) [164] reduce only 8.9% of all branch mispredictions that TAGE-SC-L incurs as they also fail to scale for large code footprints.

In this work, we focus on eliminating branch mispredictions with Whisper—a profile-guided technique that identifies branch instructions causing frequent mispredictions, correlates their direction with many prior branch directions (*i.e.*, history), and efficiently encodes this correlation using Boolean formulas. In particular, Whisper introduces three novel techniques to improve profile-guided branch prediction and reduces 16.8% of all mispredictions by leveraging (1) *hashed history correlation*, (2) *randomized Boolean formula testing*, and (3) an extension of ROMBF [164] with Boolean *Implication and Converse Non-Implication* operations.

Hashed history correlation. Prior profile-guided techniques either consider extremely long histories requiring kilobytes of metadata storage per static branch [405], or utilize short (typically 4 or 8) fixed-length histories that fail to predict many branches accurately [164]. To consider long histories without incurring metadata overhead, we propose hashed history correlation that correlates branch outcomes with a hash of variable-length histories in a profile-guided manner. To find the best history length for predicting a branch, Whisper considers different lengths from a geometric series and picks the length that shows the strongest correlation. Whisper converts histories of that length into a fixed-length (8-bit) hashed history and efficiently encodes this hashed history using Boolean formulas.

Randomized formula testing. Determining the optimal boolean formula for predicting the branch outcome based on an N-bit history, requires exploring a search space of size 2^{2^N}. To address this challenge, Whisper proposes randomized formula testing, a technique that only considers a random, yet uniform, subset of all prediction formulas as candidates, selecting the best formula for predicting branches. Whisper finds near optimal formulas, comparable to exhaustive exploration (88.3% on average) while considering only 0.1% of all possible prediction formulas.

Implication and Converse Non-Implication operations. Besides reducing the search space of Boolean formulas, Whisper also improves their prediction accuracy. In particular, Whisper introduces Implication and Converse Non-Implication that improve prediction accuracy over ROMBF by 1.5% while maintaining the low storage cost of ROMBF.

Whisper enables these three contributions with a novel PGO technique. In particular, it collects the execution profile of data center applications in production using efficient hardware support [1, 7] and then performs an offline branch analysis. The analysis yields optimized ROMBF enabling the injection of `brhint` instructions for branches that cause frequent mispredictions.

The `brhint` instruction efficiently encodes precise history lengths, a Boolean formula to differentiate taken histories from not-taken histories (and vice versa), and a pointer to the corresponding branch instruction. Using the state-of-the-art profile-guided correlation algorithm [199, 202, 195], Whisper inserts the `brhint` instruction in a suitable predecessor of the branch at link time to ensure hint timeliness. At run time, Whisper utilizes the hint of a corresponding branch instruction to compare the hashed dynamic history against the Boolean formula for predicting the branch outcome. Thus, Whisper leverages hardware/software co-design to eliminate data center applications' branch mispredictions in a profile-guided manner.

We evaluate Whisper for 12 popular data center applications that suffer from frequent frontend and misprediction stalls and show that, on average, Whisper eliminates 16.8% of all branch mispredictions over the 64KB state-of-the-art TAGE-SC-L [317] baseline. Due to this 1.7%-32.4% reduction in mispredictions, Whisper achieves an average speedup of 2.8% (0.4%-4.6%) for data center applications. Compared to state-of-the-art profile-guided branch prediction mechanisms [405, 164], Whisper achieves 1.1% greater speedup while reducing 7.9% more branch mispredictions. By injecting `brhint` instructions, Whisper increases the code footprint by 11.4% and executes 9.8% extra dynamic instructions.

We make the following contributions:

- An extensive investigation of branch instructions' behavior in data center applications demonstrating that large code footprints of these applications trigger frequent branch mispredictions, significantly limiting the overall performance.
- Whisper: a novel profile-guided mechanism to eliminate branch mispredictions in data center applications. Whisper correlates a given branch's direction with many prior branch directions, efficiently encodes this correlation using Boolean formulas, and improves the overall efficacy of branch prediction.
- A comprehensive evaluation of Whisper for 12 data center applications that shows that Whisper can eliminate costly branch mispredictions (16.8% on average) and achieve substantial performance benefits (2.8% on average).

7.2 Branch Prediction Challenges for Data Center Applications

In this section, we thoroughly investigate the behavior of branch instructions from 12 real-world data center applications to show that branch mispredictions significantly limit their overall performance. Then, we explain why state-of-the-art branch predictors fail to eliminate these branch mispredictions. Finally, we provide valuable insights on how to overcome branch mispredictions

Table 7.1: Data center applications and workloads we study.

Applications	Workloads
MySQL [8]	Different TPC-C queries [84]
PostgreSQL [10]	Different pgbench queries [9]
Clang [5]	Building LLVM [216]
Python [17]	pyperformance benchmarks [13]
Finagle-chirper [15] Finagle-http [15]	Java Renaissance benchmark suite [294]
Cassandra [2] Kafka [3] Tomcat [4]	Java DaCapo benchmark suite [51]
Drupal [378] Wordpress [383] Mediawiki [380]	Facebook's OSS-performance suite [19]

Table 7.2: Simulator parameters

Parameter	Value
CPU	3.2GHz, 6-wide OOO, 24-entry FTQ, 224-entry ROB, 97-entry RS
Branch prediction unit	64KB TAGE-SC-L [317] (up to 12-instruction), 8192-entry 4-way BTB, 32-entry RAS, 4096-entry IBTB
Caches	32KB 8-way L1i, 32KB 8-way L1d, 1MB 16-way unified L2, 10MB 20-way shared L3 per socket

for data center applications.

7.2.1 Experimental methodology

Data center applications. Recent work from Facebook and Google reports that their widely-deployed data center applications exhibit multi-megabyte code footprints [279, 278, 41, 179] and consequently lose more than 15% of all pipeline slots directly due to branch mispredictions [344, 39]. Due to large instruction footprints, these applications also lose more than 29% of all pipeline slots due to frontend stalls [179, 39, 41, 344]. Accurate and timely branch predictions can effectively hide a large fraction of these frontend stalls because of the decoupled nature [301, 302] of modern processor frontends [350, 283, 308, 129]. Since these applications and their corresponding workloads are proprietary, we use open-source applications and workloads used by prior work [411, 217, 199, 202, 195, 362, 259, 278, 279] with large code footprints that similarly cause frequent branch mispredictions and frontend stalls. We describe these data center applications and their workloads in Table 7.1.

Trace collection and simulation parameters. We collect these applications' traces using Intel PT [1] and simulate these traces using the Scarab [14] simulator. Table 7.2 lists different simulation parameters that resemble a recent state-of-the-art industry baseline [150, 151].

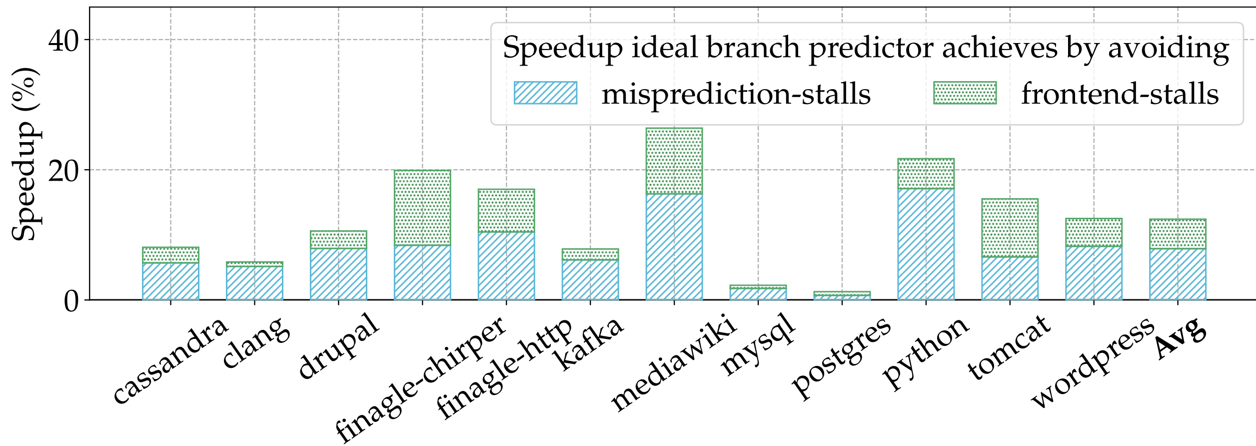


Figure 7.1: Data center application limit study: an ideal branch predictor achieves an average IPC speedup of 12.4% (1.3%-26.4%) over the state-of-the-art 64KB TAGE-SC-L baseline.

7.2.2 Why is branch prediction important for data center applications?

To understand the importance of the branch prediction mechanism for data center applications, we perform a limit study to measure the maximum performance benefits of an ideal branch direction predictor over the state-of-the-art 64KB TAGE-SC-L [317] predictor. For this ideal branch predictor, only the prediction direction is ideal, *i.e.*, it always predicts taken and not-taken branches correctly. In Fig. 7.1, we show that the ideal branch direction predictor achieves an average Instructions Per Cycle (IPC) speedup of 12.4% (1.3%-26.4%) over the state-of-the-art TAGE-SC-L branch predictor.

To understand the reason behind this significant performance gap, we break down the speedup into two categories: (1) speedup due to avoiding branch misprediction stalls (*i.e.*, pipeline squashes [212]) and (2) speedup due to avoiding frontend stalls by performing FDIP [301, 302]. For traditional benchmarks (*e.g.*, SPEC2017), avoiding misprediction stalls is the primary benefit of ideal branch prediction. However, for data center applications, eliminating branch mispredictions is also important as it reduces I-cache misses through FDIP.

As also shown in Fig. 7.1, among the 12.4% mean IPC speedup provided by the ideal branch predictor, an average IPC speedup of 7.9% (0.7%-17.1%) is provided by eliminating all branch misprediction stalls for these applications. On top of that, the ideal branch predictor achieves an additional 4.5% speedup on average (0.5%-11.5%) by eliminating frontend stalls (I-cache misses) for these applications. Therefore, eliminating branch mispredictions is extremely critical for data center applications.

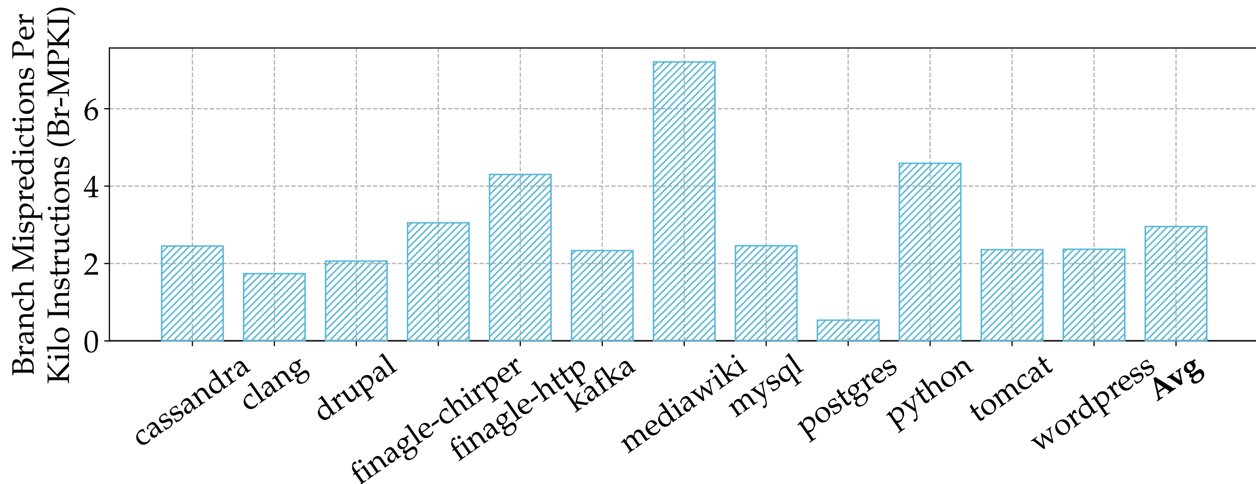


Figure 7.2: Branch Mispredictions Per Kilo Instructions (branch-MPKI) for 12 data center applications: 64KB TAGE-SC-L experiences an average branch-MPKI of 3.0 (0.5-7.2) for these applications.

7.2.3 Why does the state-of-the-art TAGE-SC-L branch predictor fall short?

We now investigate why the state-of-the-art TAGE-SC-L branch predictor is insufficient for data center applications with large code footprints.

Fig. 7.2 shows the branch-MPKI of 64KB TAGE-SC-L across all 12 data center applications. While measuring the branch-MPKI, we only consider mispredictions caused by conditional branch instructions, following the methodology of 5th Championship Branch Prediction (CBP-5) [18]. As shown in Fig. 7.2, TAGE-SC-L exhibits a branch-MPKI in the range of 0.5-7.2 (3.0 on average) for the analyzed data center applications. To understand the reason behind these frequent branch mispredictions, we categorize all branch mispredictions TAGE-SC-L induces among four different classes: (1) Compulsory mispredictions, (2) Capacity mispredictions, (3) Conflict mispredictions, and (4) Conditional-on-data mispredictions. We perform this classification by analyzing consecutive accesses of a branch substream—the combination [400, 218, 250, 267, 277, 330, 397] of branch instruction's Program Counter (PC) and history of different lengths.

Compulsory [142, 351, 255] mispredictions occur when TAGE-SC-L predicts a branch for the first time and the predicted direction does not match with the true direction. *Capacity* [142, 351, 255] mispredictions occur when the reuse distance [156, 97] of a branch is too large so that the substream is evicted from the TAGE-SC-L tables. *Conflict* [142, 351, 255] mispredictions occur when the associativity or the replacement mechanism for TAGE-SC-L tables is not effective enough to retain the branch substream between two consecutive accesses. *Conditional-*

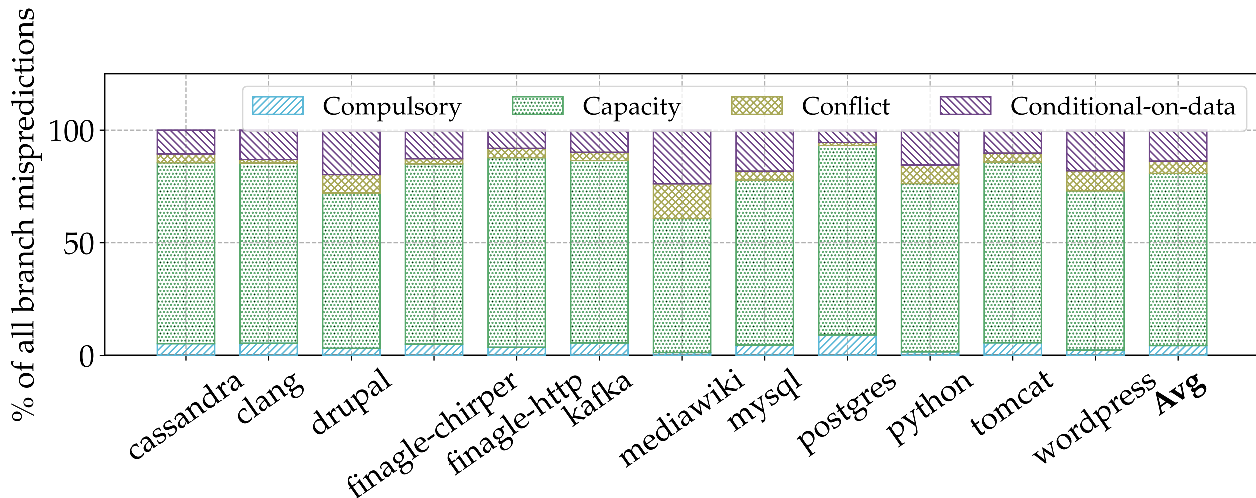


Figure 7.3: Breakdown of all branch mispredictions among 4 different classes [142, 351, 255]: data center applications suffer from frequent branch mispredictions primarily (76.4% of all mispredictions) due to capacity issues.

on-data mispredictions occur when the branch's direction depends on data values and does not correlate with prior history. Consequently, history-based predictors like TAGE-SC-L cannot achieve high prediction accuracy for such branches [111].

Fig. 7.3 shows the breakdown of all branch mispredictions TAGE-SC-L incurs across different categories. As shown, the majority of these mispredictions occur due to capacity reasons (on average 76.4%).

This result reveals that the working set size of branch substreams for data center applications is significantly larger than the capacity of even the 64KB state-of-the-art TAGE-SC-L branch predictor. Furthermore, this characterization confirms that large instruction footprints of modern data center applications put extreme pressure on branch predictors in addition to the instruction cache, instruction translation lookaside buffer, and branch target buffer as prior works have observed [112, 179, 65, 39, 274, 344, 41, 411, 278, 217, 199, 202, 195, 362, 259, 279, 275].

7.2.4 Why do existing profile-guided techniques fall short?

We now investigate the degree to which prior profile-guided branch prediction techniques solve the large branch footprint problem of modern data center applications. We primarily present the analysis for BranchNet [405], the most recent profile-guided branch prediction technique, and ROMBF [164], the most effective profile-guided technique for data center applications in our study. These techniques are hybrid in nature as they use profile-guided techniques for hard-to-predict branches and use TAGE-SC-L for remaining branches.

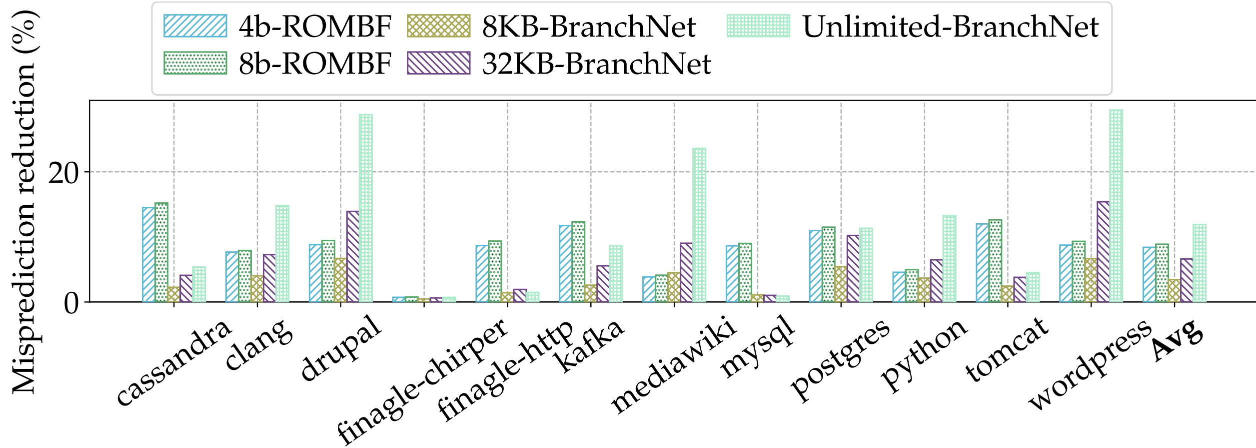


Figure 7.4: Performance of prior profile-guided branch prediction techniques [164, 405] over the 64KB TAGE-SC-L baseline: these techniques reduce only 3.4%-8.9% of all branch mispredictions TAGE-SC-L incurs. Even with unlimited storage, this impractical variant of BranchNet [405] achieves an average misprediction reduction of only 11.9%.

BranchNet. BranchNet [405] deploys Convolutional Neural Networks (CNNs) for hard-to-predict branches together with traditional online branch predictors (*e.g.*, TAGE-SC-L). To train CNNs for these branches, BranchNet leverages offline profiles from multiple application inputs. At run time, TAGE-SC-L makes predictions for the vast majority of branches while CNNs predict the few hard-to-predict branches. Based on metadata storage, BranchNet also proposes different variants of CNNs: (1) 8KB-BranchNet and (2) 32KB-BranchNet. To understand the potential of CNNs for predicting branches, we also study BranchNet with no storage restrictions, unlimited-BranchNet.

Read-Once Monotone Boolean Formulas (ROMBF). Prior work [164] utilizes Boolean formulas to predict branch outcomes based on history. In particular, every branch outcome in the history represents a Boolean variable that is combined using logical operations (*e.g.*, `and`, `or`) to predict a branch's direction. Branch prediction using Boolean formulas faces two key challenges. First, to determine the optimal Boolean formula that provides the best prediction accuracy for a history length of N, the approach has to explore 2^{2^N} all possible formulas. Second, to encode the Boolean formula, the approach requires 2^N-bit storage. Prior work [164] addresses only the second challenge by using a subset of Boolean formulas where every variable appears exactly once and by allowing only two logical operations `and` and `or`. Consequently, prior work [164] encodes a ROMBF of N variables using only $N - 1$ bits. Using such a compact encoding, prior work annotates branch instructions with N-bit hints to make branch predictions based on the outcome of the last N branches. The study also proposes different variants of ROMBF (4-bit and 8-bit) for different values of N. For brevity, we refer to this prior work [164] as ROMBF.

To assess the potential of these existing profile-guided branch prediction mechanisms, we eval-

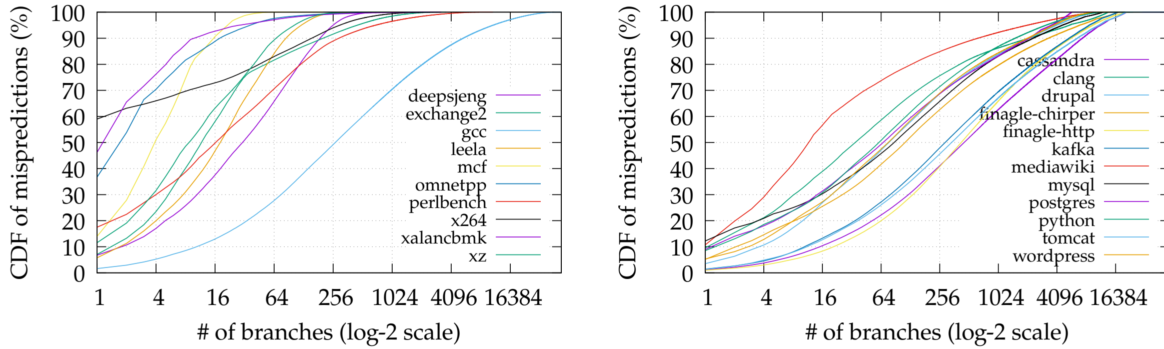


Figure 7.5: The distribution of all branch mispredictions across different branch instructions using TAGE-SC-L for SPEC2017 integer benchmarks (left) and data center applications (right). In general, SPEC benchmarks satisfy BranchNet's [405] assumption as only a top-few (*e.g.*, 50) branch instructions cause the majority (*e.g.*, $> 60\%$) of all mispredictions. Data center applications, however, do not satisfy this assumption as mispredictions are distributed across thousands of different branches.

uate BranchNet and ROMBF over the 64KB TAGE-SC-L baseline. As shown in Fig. 7.4, data center applications do not significantly benefit from these existing mechanisms. Specifically, the state-of-the-art profile-guided technique, BranchNet, reduces only 3.4% and 6.6% of all branch mispredictions with 8KB and 32KB metadata storage. Even with the unlimited metadata storage, BranchNet only avoids 11.9% of all branch mispredictions. On the other hand, ROMBF reduces 8.4% and 8.9% of all branch mispredictions using 4-bit and 8-bit formulas. Next, we investigate the performance of these prior profile-guided techniques to understand why they fail to avoid so many branch mispredictions.

BranchNet employs CNNs to predict hard-to-predict branches assuming that only a few static branches disproportionately cause the vast majority of all mispredictions for an application. For example, as shown in Fig. 7.5, the top 50 static branches experience more than 60% of all mispredictions for SPEC2017 integer speed benchmarks (*e.g.*, `leela`, `xz`, `omnetpp`, `deepsjeng`, and `mcf`). Consequently, for these benchmarks, BranchNet can reduce 12.6%-34% of all mispredictions by allocating 256B-2KB metadata storage for each of these branches' CNNs. However, as also shown in Fig. 7.5, mispredictions for data center applications and `gcc` (from SPEC) are more uniformly distributed across many static branches. Consequently, for these applications, even unlimited-BranchNet can only avoid 11.9% of all mispredictions while using 2KB CNNs for each static branch.

ROMBF predicts a branch by applying an N-bit formula to the last N branch outcomes. For example, 4-bit and 8-bit formulas can predict branches based on only the last 4 and 8 branch outcomes. As shown in Fig. 7.6, most branches in our data center applications correlate with branch histories of size 32-1024 and, consequently, 4-bit and 8-bit formulas are insufficient. As

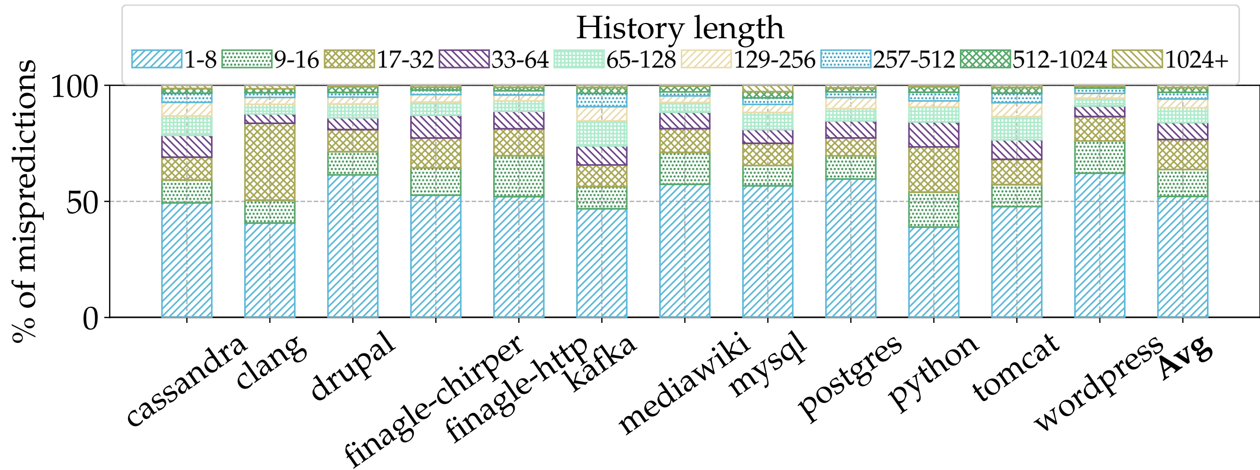


Figure 7.6: Distributions of all branch mispredictions among different history lengths. Predicting a branch requires correlating its direction with even 1024 prior branch outcomes.

ROMBF requires N-bit hints to consider N-bit histories, it does not scale well for long branch histories.

Furthermore, ROMBF only considers `and` and `or` operators to compute Boolean formulas along with contradiction (*i.e.*, never taken) and tautology (*i.e.*, always taken). This limitation assumes that these formulas can encode relevant histories for the large majority of branches without any quantitative insight. We characterize the implications of this assumption in Fig. 7.7 by showing the distribution of all branches among formulas using contradiction, tautology, `and`, `or`, implication, and converse non-implication. As shown, while formulas using `and` (28.9%) and `or` (5.3%) operations represent histories of a significant number of branches, formulas using implication (8.8%) and converse non-implication (9.2%) operations also encode histories of a large number of branches. Consequently, ROMBF can not avoid mispredictions for these branches.

In §7.5.2 (Fig. 7.16) we will show that BranchNet requires orders of magnitude higher training time than ROMBF, while Whisper outperforms both approaches. Next, we use the insights from these characterizations to design Whisper, our profile-guided technique to eliminate branch mispredictions for data center applications.

7.3 Design of WHISPER

Our investigation reveals that ideal branch prediction significantly improves the performance of data center applications as their large branch footprints exhaust 64KB TAGE-SC-L [317]. State-of-the-art profile-guided mechanisms [405, 164] also fail to eliminate a large majority of branch mispredictions for these applications. We propose Whisper, a combination of three novel profile-

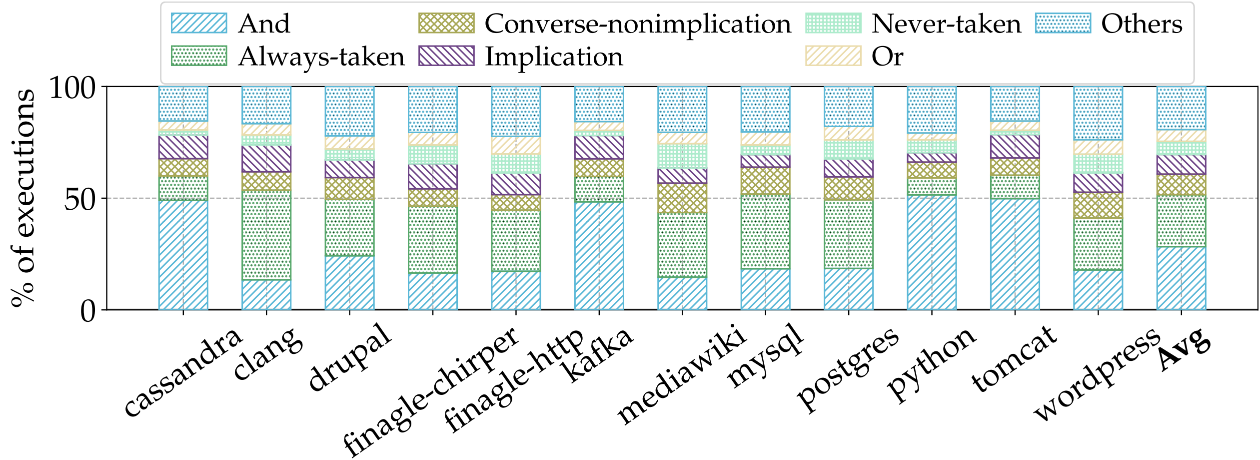


Figure 7.7: Distributions of branch executions among different logical operations used in the Boolean formula to predict a branch. And (28.9%), always-taken (23.3%), converse non-implication (9.2%), implication (8.8%), never-taken (5.9%), and Or (5.3%) operations together can predict more than 80% of all branch executions.

guided techniques to improve branch prediction. Whisper introduces hashed history correlation to predict branches that correlate with long variable-length histories. Furthermore, Whisper proposes randomized formula testing to reduce the massive offline training time of existing profile-guided branch prediction techniques [405, 164] without affecting the prediction accuracy. Finally, Whisper extends ROMBF by including Implication and Converse Non-Implication operations to predict branches accurately.

Whisper leverages profile-guided analysis at link time to correlate branches with previous branch outcomes using efficient hardware-based control flow tracing support such as Intel PT [1] and LBR [7]. Next, Whisper maps values of variable-length histories corresponding to different branch outcomes into fixed-length hashed values and encodes these hashed values using an extended ROMBF formula. To pick the formula for any given branch, Whisper considers a randomized subset of all formulas and selects the formula yielding the fewest mispredictions for all hashed histories of the branch. Whisper annotates every hard-to-predict branch with its corresponding formula to provide the branch predictor in hardware with the following information: (1) how many prior branches in the global history are relevant for predicting the current branch, and (2) how the outcome of these prior branches need to be combined to compute the direction of the current branch. Whisper introduces minor hardware modifications to match the dynamic history with an annotated formula, predicting the corresponding branch outcome. We describe Whisper's in-depth usage model in §7.4. Now, we discuss the novel techniques Whisper proposes along with its micro-architectural modifications in greater detail.

7.3.1 Hashed history correlation

As shown in §7.2.4, the computational complexity of learning optimal ROMBF and their storage overhead increases linearly with the number of considered variables. In the context of branch prediction, these variables are previous branch directions, *i.e.*, the global branch history leading into a branch. As a result, prior work [164] is limited in accuracy by only considering short histories. To address this challenge, Whisper introduces hashed history correlation, providing three key capabilities: (1) an efficient encoding scheme of large and variable-length histories, (2) a technique to correlate a subset of the branch history with a specific branch outcome, and (3) a mechanism to represent different history values utilizing a single formula.

History hashing. Whisper introduces history hashing that converts the history of any arbitrary length into a fixed length. For example, Whisper transforms the 64-bit history (*i.e.*, the outcome of the most recent 64 branches) into a 16-bit hashed history by dividing the 64-bit value into four 16-bit chunks and applying logical operations (*e.g.*, `and`, `or`, `xor`) to these 16-bit chunks. We empirically study the sensitivity of Whisper's hashing mechanism for different hashed lengths and different logical operations to find that the 8-bit hash and `xor` operations provide a good trade-off between instruction footprint overhead and prediction accuracy. As branch predictors used in today's hardware already use a similar hashing mechanism [24], Whisper does not introduce significant micro-architectural modifications to perform history hashing.

History correlation. Directions for different branches correlate with prior histories of different lengths. Some branches correlate with the outcome of only the most recent branches while other branches correlate with the outcome of relatively older branches. Whisper addresses this challenge by considering various history lengths for each static branch and selecting the length that provides the highest accuracy for that branch using profile samples.

Whisper's hashed history correlation technique requires three parameters: (1) the minimum history length a, (2) the maximum history length N, and (3) the number of different history lengths m. To find the best history length for a branch, Whisper analyzes all execution samples, referred to as substreams, for that branch using an application profile collected via efficient hardware support (Intel PT [1] and LBR [7]). Each substream contains two components: (1) the actual direction of the branch execution and (2) the directions of the most recent N branches before that branch. Using these scenarios, Whisper determines the best history length and formula for a given branch by evaluating a list of potential history lengths.

For each branch, Whisper considers different history lengths that follow a geometric series [316], up to the m-th term, starting with the minimum history length, a, *i.e.*, $a, ar, ar^2, \cdots, ar^{m-1}$, where $r = (\frac{N}{a})^{(\frac{1}{m-1})}$. At each history length in the series, Whisper encodes the branch history up to this length. As described above, to minimize storage costs, Whisper does

not evaluate raw, full-length histories. Instead, Whisper operates on hashed histories, allowing it to compare histories of different original lengths. Next, Whisper determines a Boolean formula that best fits the substream (see §7.3.2). This is done by using the total number of taken and not-taken counts for the hashed partial history across all samples for that branch. Then, Whisper counts the total number of mispredictions that the current history length and formula incur. If there is a history length that results in the fewest mispredictions for that branch, then that history length is considered the best and used later at run time. If none of these history lengths improve accuracy when compared to the profiled results, then Whisper indicates that the given branch should be predicted in a purely dynamic manner (*i.e.*, using the underlying branch predictor). Additionally, we empirically study Whisper's sensitivity to different parameters (a, N, and m) and observe that the values $a = 8$, $N = 1024$, and $m = 16$ work well.

History representation. The primary goal of Whisper's profile-guided analysis is to annotate a static branch with a Boolean formula that efficiently encodes relevant historical branch outcomes to predict the directions of the branch accurately. As we describe in §7.2.4, in an N-bit history, where each branch can be either taken or not-taken, there exist 2^N potential branch scenarios. Whisper needs to partition these 2^N branch scenarios into two groups using a Boolean formula, where one group reflects the scenarios in which the branch is taken and the other group where the branch is not taken. To achieve this goal, Whisper considers several Boolean formulas for each static branch and selects the Boolean formula that can predict the branch with the highest accuracy based on the collected profile. Algorithm 3 shows a simplified version of the technique Whisper utilizes to find the best formula for representing each branch's history.

Algorithm 3 takes two hash tables, T and NT as inputs. They contain the hashed history as keys and the number of profile samples as values. T and NT denote taken and not-taken samples respectively. As output, Algorithm 3 generates the Boolean formula, f, that incurs the minimum number of mispredictions, m'.

As shown in Algorithm 3, Whisper initializes the minimum number of mispredictions, m', with the value ∞ (Line 1) and the best Boolean formula, f, with a default value of $\emptyset$ (Line 2). Next, Whisper generates the list of all Boolean formulas that will be considered as candidates for predicting the branch (Line 3). We will later (§7.3.2 and §7.3.3) describe how Whisper finds only a subset of Boolean formulas that approximates the full potential of all Boolean formulas with high accuracy and efficiency.

For each formula f', Whisper initializes the total number of mispredictions the formula sustains, t, as 0 (Line 5). Next, Whisper iterates over each key-value pair of T (Lines 6-8) and NT (Lines 9-11) to calculate the value of t. Since each key k denotes the hashed history, Whisper first determines whether k satisfies the Boolean formula f' (Line 7 and 10 for T and NT respectively). For taken samples (T), if k does not satisfy f', predicting the branch using f' will result

Algorithm 3 Finding the best Boolean formula to differentiate taken histories from not-taken histories.

FIND-BOOLEAN-FORMULA (T, NT)

Input: T and NT contain different hashed history as keys and the number of profile samples as values. T and NT denote taken and not-taken samples respectively.
Output: The Boolean formula, f which incurs the minimum number of mispredictions, m'

```
1:  m' ← ∞
2:  f ← ∅
3:  F ← List-of-Considered-Formulas ()
4:  for each f' ∈ F do
5:      t ← 0
6:      for each k ∈ T.keys do
7:          if satisfy (k, f') ≠ 1 then
8:              t ← t + T[k]
9:      for each k ∈ NT.keys do
10:         if satisfy (k, f') = 1 then
11:             t ← t + NT[k]
12:     if t < m' then
13:         f ← f'
14:         m' ← t
15: return(f, m')
```

in mispredictions. Therefore, Whisper adds the corresponding number of profile samples, $T[k]$, to t (Line 8). Similarly, for not-taken samples (NT), if k satisfies f', predicting the branch using f' will also result in mispredictions, so Whisper also adds the corresponding number of profile samples, $NT[k]$, to t (Line 11). Thus, Whisper counts the total number of mispredictions f' incurred for all profile samples.

Finally, Whisper compares t with m' to decide whether the current formula, f' causes the minimum number of mispredictions (Line 12). If t is smaller than m', Whisper updates f and m' with the values f' and t correspondingly (Lines 13-14). Whisper produces the final values of f and m' as output after iterating over all formulas from the subset of considered Boolean formulas (Line 15). Next, we explain how Whisper efficiently generates only a subset of all Boolean formulas that effectively achieves the high accuracy of considering all Boolean formulas.

7.3.2 Randomized formula testing

As we discuss in §7.2.4, any N-bit variable can take 2^N different values. Therefore, finding the best formula that predicts a branch with the least number of mispredictions requires exhaustively searching the search space of all 2^{2^N} different formulas. For example, predicting a branch based on

the outcome of the last 4 branches will require testing 65536 ($= 2^{2^4}$) different possible formulas. While testing one formula does not depend on the outcome of a different formula, *i.e.*, checking all formulas is embarrassingly parallelizable, it still requires a large amount of computational operations. Whisper leverages randomized formula testing to reduce this exponential search space.

To perform randomized formula testing, Whisper first generates a random permutation of all formulas using the Fisher-Yates shuffle algorithm [116, 101]. The Fisher-Yates shuffle algorithm ensures that Whisper generates the random order only once and reuses this order for all different branches. For each branch, Whisper selects only a fraction of all formulas to consider as potential candidates to predict the branch. Among these selected candidates, Whisper picks the best formula using Algorithm 3. We investigate the implications of randomized formula testing to the fraction of all formulas tested in §7.5.2 (Fig. 7.15) and show that Whisper achieves comparable performance to exhaustive search (88.3%) even after checking only 0.1% of all Boolean formulas.

7.3.3 Implication and Converse Non-Implication

As discussed in §7.2.4, when considering arbitrary Boolean formulas for N-bit variables, we need to evaluate 2^{2^N} formulas and also need 2^N-bits of storage for tagging each hard-to-predict branch. As accurate branch prediction often requires significantly larger histories, prior work [164] proposed ROMBF to reduce the storage overheads of these formulas to N-bits. Unfortunately, considering every variable only once leads to sub-optimal Boolean formulas as it is impossible to represent formulas where variables appear twice (*e.g.*, $(a\&\&b)||(!a\&\&c)$). Whisper addresses the reduced accuracy provided by ROMBF by introducing additional operations such as contradiction, tautology, `and`, `or`, implication, and converse non-implication. This approach enables more powerful Boolean formulas, improving branch prediction accuracy while increasing storage only linearly.In particular, Whisper requires $log_2(op) * hash(n)$-bits for each formula, where op represents the number of supported operations and n denotes the number of branches considered in the history. As discussed in §7.3.1, Whisper also utilizes hashing to represent longer histories of size n because fewer bits are produced by the hash function.

Micro-architectural implementation. Adding Implication and Converse Non-Implication requires minor micro-architectural modifications to the original hardware implementation of ROMBF [164]. Fig. 7.8 shows an implementation for predicting the branch direction based on the outcome of the last two branches ($N = 2$). For two data inputs (b_0 and b_1), Whisper requires three control inputs (Ⓞ$_0$, Ⓞ$_1$, and Ⓘ). As a single unit, Whisper produces the outcome of all four logical operations using b_0 and b_1. Then, Whisper selects the output based on the two control inputs (Ⓞ$_0$ and Ⓞ$_1$) using a 4×1 multiplexer. Finally, Whisper selects either the output of the multiplexer or its inverted value based on the remaining control input, Ⓘ using another 2×1

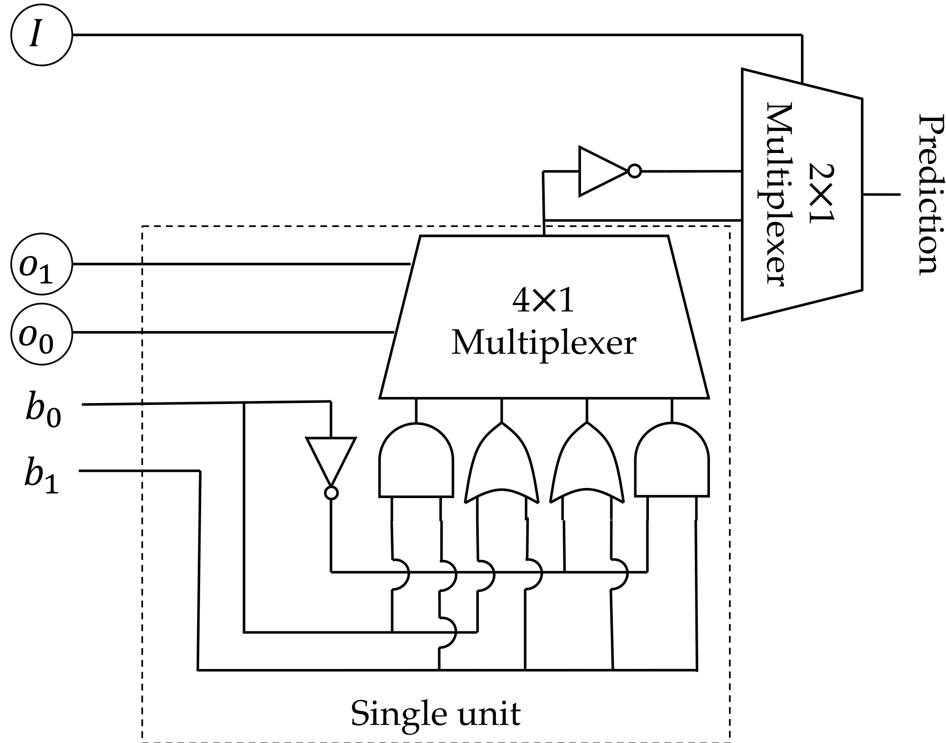


Figure 7.8: Micro-architecture of the Read-Once Monotone Boolean Formulas Whisper extends with Implication and Converse Non-Implication. It shows the single unit to predict a branch based on the outcome of the last 2 branches.

multiplexer. Next, we describe how Whisper combines multiple single units in general ($N > 2$).

Fig. 7.9 shows the micro-architectural requirements of Whisper's mechanism to predict a branch based on the direction of the last 8 branches. Whisper uses four single units that operate on the outcomes of prior branches, $b_0, b_1, \cdots, b_7$. Then, Whisper uses outputs of these single units as inputs to two single units in the next layer. Next, Whisper uses the output of these two single units as inputs to a single unit in the last layer. All of these single units at different layers require 14 ($2 \times (8-1)$=2×7) control inputs, O_0 to O_{13}. Finally, Whisper uses a 2×1 multiplexer to select either the last layer's output or its inverted value based on I.

As shown in Fig. 7.9, Whisper performs most of the Boolean operations at a single layer in parallel. The longest delay Whisper incurs is due to 3 sequential single units at different layers following the final step that uses the 2×1 multiplexer. Every single unit has a maximum delay of 5 logic gates: Not gate, And/Or gate, and three gates for the 4×1 multiplexer. The final step's maximum delay is 4 logic gates: Not gate and three gates for the 2×1 multiplexer. The hashing operation does not incur any extra overhead as existing processors already perform similar hashing operations [24]. Thus, Whisper incurs a maximum delay of only 19 logic gates. Even if Whisper can not compute this entire logic in a single cycle, Whisper can easily pipeline these operations,

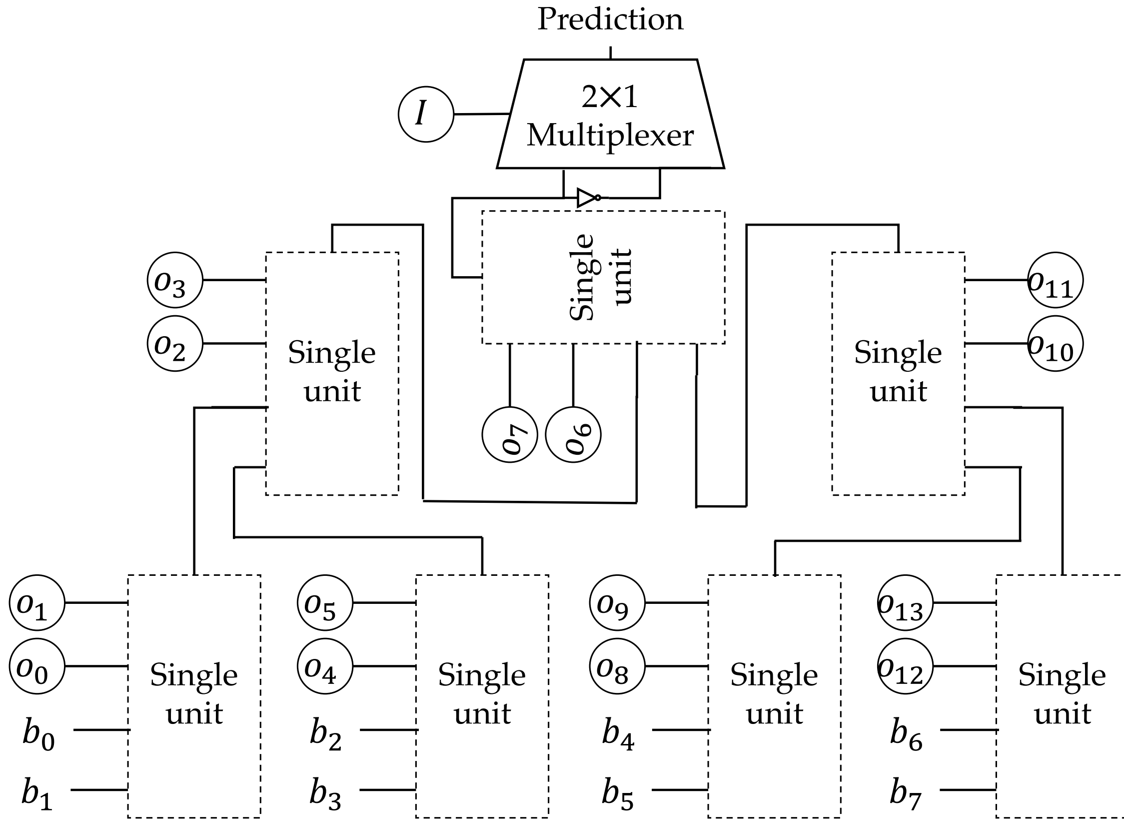


Figure 7.9: Micro-architecture showing how Whisper combines multiple single units in general. This shows how Whisper predicts a branch based on the outcome of the last 8 branches.

e.g., by registering the results of the first ten operations in one cycle and performing the last nine operations in the next cycle. In any event, Whisper generates its prediction in parallel with TAGE-SC-L, whose logical depth and complexity with hashed SRAM table lookups, tag comparisons, and adder tree for the SC component exceed Whisper's complexity.

7.4 Usage Model

We show the high-level usage model of Whisper in Fig. 7.10. Whisper collects data center applications' execution profiles in production and analyzes these profiles offline to inject branch hint instructions.

Run-time profiling. First, Whisper collects the execution trace of branch instructions for data center applications in production (step ①) using Intel PT [1] and LBR [7]. Similar to recent work [195, 202, 199, 41], Whisper leverages Intel PT and LBR as they are widely adopted in today's data centers [86, 105, 65, 278, 279]. Intel PT captures the trace of dynamically executed branch instructions with low overhead (only up to 1% [184, 185, 188, 413]). As shown in Fig. 7.10,

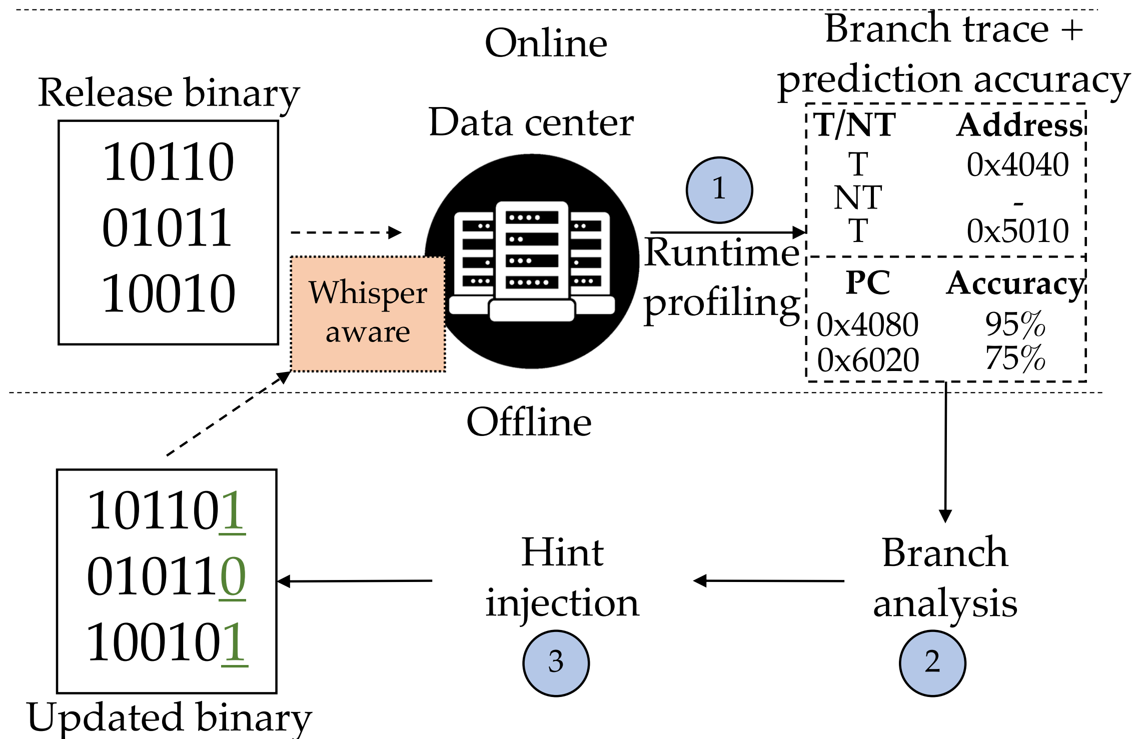


Figure 7.10: Whisper's usage model.

this trace contains a branch direction (taken, T or not-taken, NT) for each branch instruction along with the next instruction's address when an indirect branch is taken. Intel LBR provides Whisper with the prediction accuracy of each dynamically executed branch instruction for the underlying branch predictor. Similar to PT, LBR also incurs minimal overhead [196, 157].

Branch analysis. Next, in step ②, Whisper analyzes the in-production execution trace of branch instructions. For a static branch instruction, Whisper considers all of its dynamic executions and the profiled processor's prediction accuracy of the branch to find the best history length using the hashed history correlation technique (§7.3.1). Also, Whisper determines the best history length for a branch only if Boolean formula-based prediction achieves better accuracy than the profiled processor's predictor for the branch. For such branches, Whisper injects an extra instruction per branch in the binary specifying hint to predict the branch.

Hint injection. Whisper's offline analysis identifies branches for which history-based Boolean formulas achieve better prediction accuracy than the profiled processor's predictor. Whisper injects a hint instruction, `brhint`, for each of these branches. A `brhint` instruction includes 4 specific components as we show in Fig. 7.11.

The first component specifies the `History` length from a geometric series. As described in §7.3.1, Whisper uses the geometric series (*i.e.*, $8, 11, 15, \cdots, 1024$) with parameter values $a = 8$, $N = 1024$, and $m = 16$ based on empirical results. The 4-bit `History` specifies which of these

History	Boolean formula	Bias	PC pointer
4-bit	15-bit	2-bit	12-bit

Figure 7.11: Different components of the `brhint` instruction Whisper proposes.

16 history lengths Whisper should use to predict the corresponding branch.

The second component specifies the 15-bit `Boolean formula` that Whisper uses to predict the branch. As described in §7.3.3, Whisper needs $2N - 1$ bits to encode a Boolean formula that predicts a branch based on the outcome of the last N branches. Consequently, the 15-bit `Boolean formula` can directly predict a branch with a history length of 8. To predict a branch with longer history lengths (*i.e.*, $11, 15, \cdots 1024$), Whisper transforms the long histories into 8-bit histories via hashing as we describe in §7.3.1.

The third component specifies the 2-bit `Bias` for always-taken and never-taken branches. The fourth component, `PC pointer`, specifies the branch instruction's program counter (PC). Whisper uses a 12-bit offset to represent branch instruction pointers as such an offset is enough to cover the vast majority ($> 80\%$) of all branch instructions [195, 337].

Instead of directly encoding the hint in the branch instruction, Whisper injects a separate `brhint` instruction for mainly two reasons. First, it avoids the instruction footprint growth for branch instructions for which Whisper does not inject any hint as these branches are predicted dynamically. Second, it also ensures hint timeliness by avoiding the requirement of pre-decoding the branch instruction. Conditional branch instructions in x86 format already support similar prefix opcodes for biased branches [83]. We extend these opcodes with additional bytes to implement the `brhint` instruction.

For a given branch, Whisper injects the corresponding `brhint` instruction in one of the predecessor basic blocks for the branch. To find the appropriate predecessor, Whisper leverages the execution trace collected in production and applies a conditional probability-based correlation algorithm [199, 202, 195].

Run-time hint usage. Whisper produces an updated binary for an application after injecting the `brhint` instructions. This updated binary is deployed in production during the next build and deployment cycle. When a data center application executes a `brhint` instruction at run time, Whisper places the corresponding four parameters in a small hint buffer. We empirically study Whisper's sensitivity to the size of this hint buffer and observe that Whisper provides high performance even with a 32-entry hint buffer.

At run time, while predicting a branch, Whisper simultaneously queries the branch predictor (*e.g.*, TAGE-SC-L) and the hint buffer. For branch PCs currently not in the hint buffer, Whisper

uses the branch predictor to predict the branch. If the hint buffer includes the branch PC, Whisper uses the hint information and the micro-architectural implementation described in §7.3.3 to predict the branch. Furthermore, Whisper ensures that the branch predictor does not allocate new entries for these branches. Thus, Whisper allows the branch predictor to allocate its storage for the remaining branches and provide better prediction accuracy.

7.5 Evaluation

7.5.1 Methodology

Data center applications and their workloads/inputs. We evaluate Whisper using 12 widely-used data center applications (as described in §7.2.1). We vary workloads/inputs for these applications by changing different database queries (*e.g.*, `oltp_read_only` vs `oltp_write_only`), different database scaling factors (*e.g.*, `100` vs `8000`), different input data and file sizes (*e.g.*, `large` vs `small`), different query mapping styles (*e.g.*, `imperative` vs `declarative`), different webpages client requests (*e.g.*, `feed=rss2` vs `p=37`), different numbers of concurrent clients (*e.g.*, `2` vs `10`), and different random number seeds (*e.g.*, `1` vs `10`). We optimize each of these applications with Whisper using the profile from one workload/input and test the performance of Whisper's optimization on a different workload/input.

Profile collection. We collect data center applications' profile using Intel LBR [7] and PT [1], and use the hardware performance event, "`br_misp_retired.conditional`" to identify branch mispredictions.

Simulation setup. We evaluate Whisper using Scarab [14] where we implement support for the `brhint` instruction and micro-architectural modifications Whisper proposes. We also modify Scarab to simulate instruction traces collected via Intel PT and evaluate Whisper by simulating 100 million representative, steady-state instructions for each application using simulation parameters listed in Table 7.2.

7.5.2 Performance analysis

Speedup. We show Whisper's speedup for 12 data center applications in Fig. 7.12. For comparison, we also show speedups that recent techniques (different variants of ROMBF [164] and BranchNet [405]) offer. To understand the limit, we also show speedups provided by the ideal branch predictor and MTAGE-SC, the best predictor in the unlimited storage category of CBP-5 [18]. As shown, Whisper provides an average speedup of 2.8% (0.4%-4.6%) that is 44.1% of the average speedup (6.3%) MTAGE-SC achieves with unlimited storage.

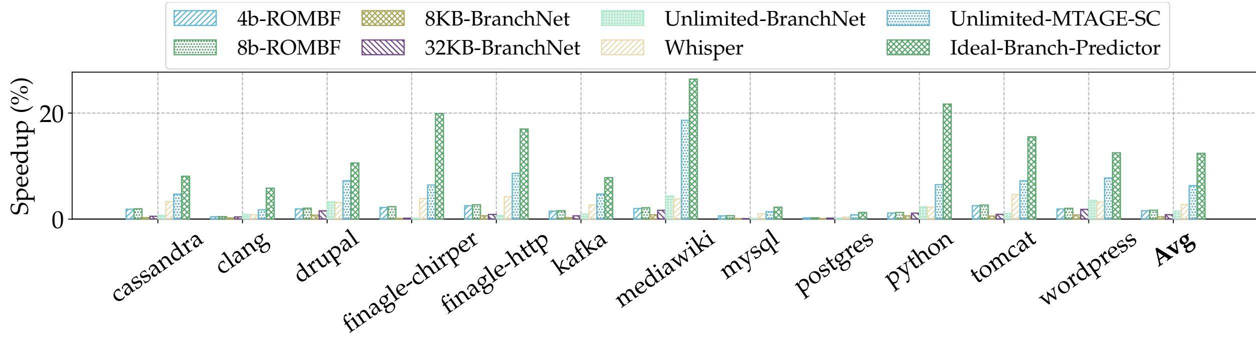


Figure 7.12: Speedup over 64KB TAGE-SC-L: Whisper achieves an average speedup of 2.8% (0.4%-4.6%) and outperforms state-of-the-art profile-guided prediction techniques [405, 164]. Whisper's speedup corresponds to 44.1% of the speedup MTAGE-SC offers with unlimited storage [318].

The speedup gap between Whisper and MTAGE-SC originates from several reasons. Whisper can not eliminate some mispredictions for previously unobserved branch instructions as Whisper optimizes applications using only one different input profile in this case. We quantify the performance implications of this input sensitivity later in this section. Furthermore, the `brhint` instructions Whisper injects incur static and dynamic instruction increases which we also quantify later in the section. Nevertheless, Whisper achieves greater speedup than prior works, ROMBF and BranchNet, as they only provide 1.7% and 0.8% on average. Furthermore, on average, Whisper provides greater speedup than BranchNet even when it leverages unlimited metadata storage. Next, we investigate how Whisper achieves this speedup by reducing a substantial amount of branch mispredictions.

Misprediction reduction. We evaluate how well Whisper reduces branch mispredictions compared to prior techniques and show the results in Fig. 7.13. As shown, on average, Whisper reduces 16.8% of all branch mispredictions (1.7%-32.4%) the TAGE-SC-L baseline incurs for these data center applications and significantly outperforms all prior mechanisms. Specifically, Whisper reduces 7.9% more mispredictions than the best performing prior technique that can be used in a practical scenario. Furthermore, Whisper outperforms the state-of-the-art, BranchNet, by 4.9% even when BranchNet uses unlimited metadata storage. This unlimited-BranchNet outperforms Whisper only for three applications (`mediawiki`, `python`, and `wordpress`) that exhibit the behavior BranchNet assumes, *i.e.*, the top-few branch instructions cause the majority of all mispredictions, as shown in Fig. 7.5. Nevertheless, Whisper eliminates more mispredictions than the practical variants (8KB and 32KB) of BranchNet even for these three applications as shown in Fig. 7.13. Next, we provide a breakdown of mispredictions Whisper eliminates among different sources of optimizations.

Breakdown of misprediction reduction. In Fig. 7.14, we show the contributions of hashed his-

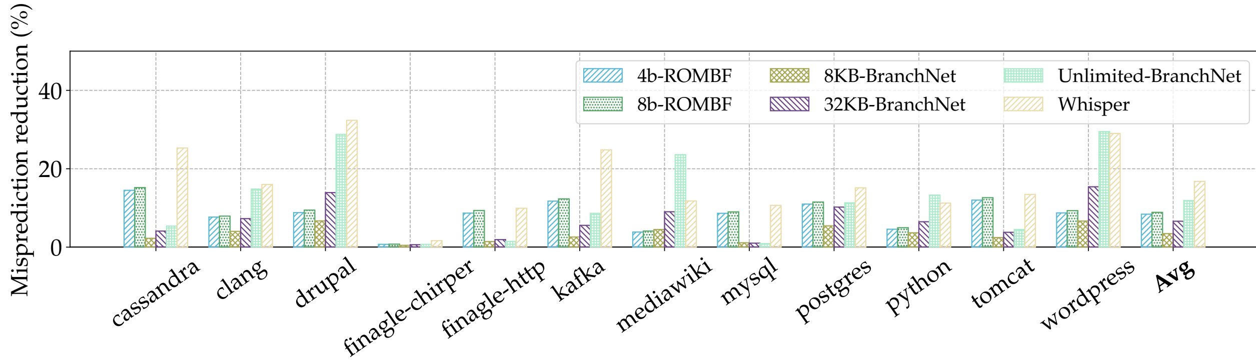


Figure 7.13: Whisper's reduction in branch mispredictions compared with BranchNet and ROMBF: Whisper eliminates 7.9% more mispredictions than the best performing realistic prior work. Whisper even removes 4.9% more mispredictions than the unlimited-BranchNet.

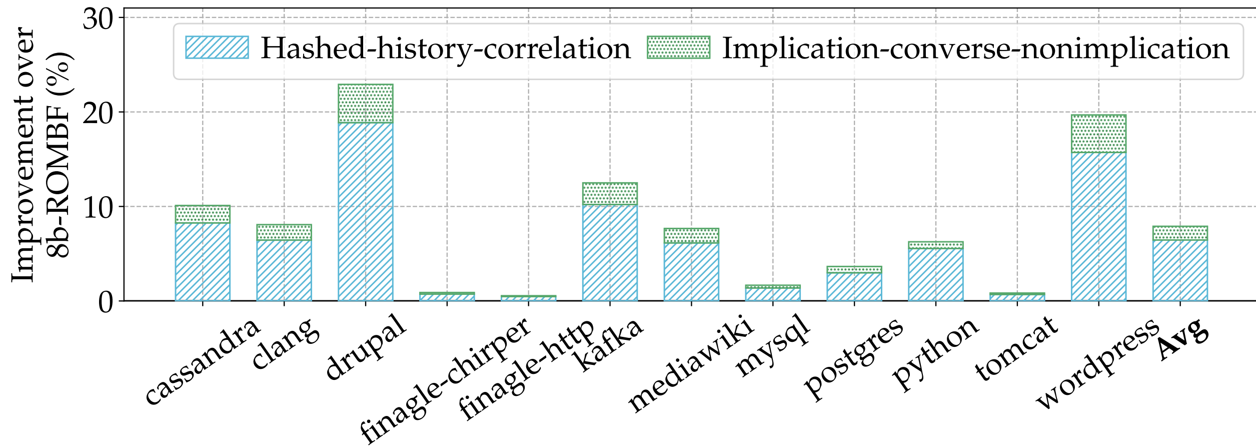


Figure 7.14: Misprediction reduction (%) achieved by hashed history correlation and Implication and Converse Non-Implication over 8-bit ROMBF: hashed history correlation reduces more branch mispredictions than Implication and Converse Non-Implication.

tory correlation and Implication and Converse Non-Implication to Whisper's overall performance. We quantify the reduction in branch mispredictions these two novel techniques offer over 8-bit ROMBF. As shown, hashed history correlation achieves an average misprediction reduction of 6.4% while Implication and Converse Non-Implication eliminate 1.5% of all mispredictions.

Implications of randomized formula testing and training time. Whisper's randomized formula testing does not eliminate any new mispredictions. Instead, randomized formula testing reduces Whisper's offline training time (*i.e.*, time to find the best Boolean formula to predict a branch) without sacrificing prediction accuracy. Fig. 7.15 shows this tradeoff between Whisper's average misprediction reduction and average training time with an increase in the percentage of formulas Whisper explores via randomized formula testing. As shown, Whisper eliminates 16.8% of all mispredictions even after exploring only 0.1% of all formulas. This reduction is comparable (88.3%

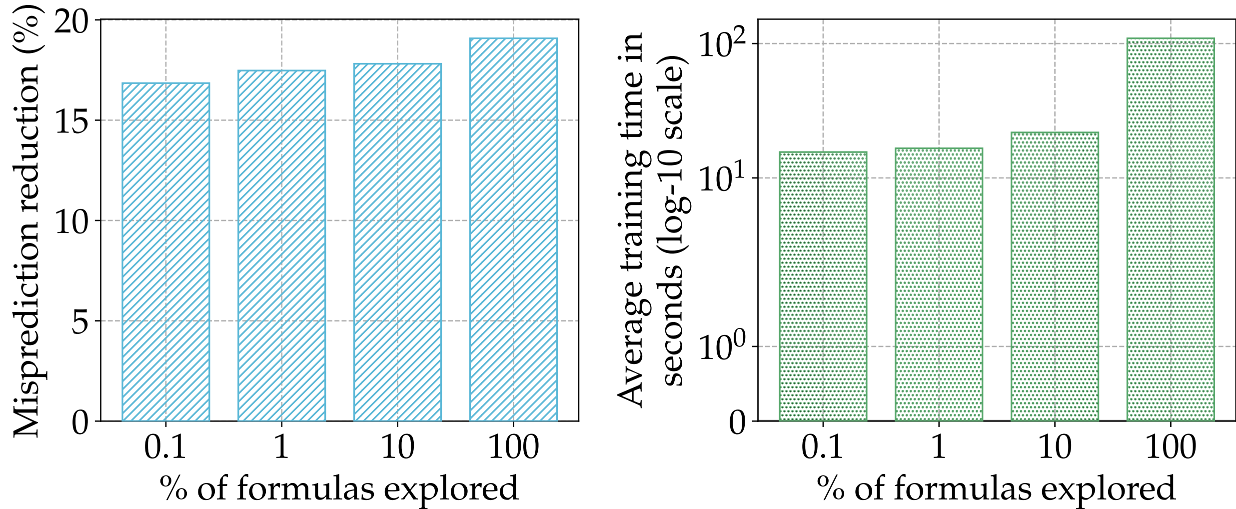


Figure 7.15: Thanks to randomized formula testing, Whisper achieves high misprediction reduction even after exploring only 0.1% of all formulas (left) while significantly reducing the training time (right, the y-axis is log-10 scale).

on average) to the reduction Whisper achieves after considering 100% of all formulas. In terms of training time, randomized formula testing is also efficient as it reduces the exploration time by an order of magnitude (Fig. 7.15). Consequently, Whisper's training time is lower than training times for 8-bit ROMBF and BranchNet (Fig. 7.16).

Performance across different workloads/inputs. As we mention in §7.5.1, we optimize data center applications with Whisper using the profile from one input and test the performance of Whisper's optimization on a different input. Now, we investigate Whisper's performance across three separate input configurations ('`#1`' to '`#3`'). We optimize each application using the training input's profile '`#0`' and measure mispredictions Whisper eliminates for different test inputs '`#1, #2, #3`'. For each input, we also measure the performance when Whisper optimizes the application with the same input's profile. As shown in Fig. 7.17, Whisper avoids 6.6% more mispredictions with input-specific profiles compared to profiles that are not input-specific.

To address this input sensitivity, prior work [405] recommended merging profiles from multiple inputs. We study the impact of merging profiles on Whisper's performance in Fig. 7.18. We compare Whisper's performance against prior works after merging profiles from different application inputs. As shown, Whisper outperforms prior techniques even for merged profiles. Furthermore, Whisper's effectiveness increases as profiles from multiple inputs are merged.

Hint overhead. Unlike BranchNet, Whisper does not incur any extra metadata overhead. Hence, Whisper's only overhead is `brhint` instructions added in the program binary and executed at run time. We estimate the static and dynamic overhead of these `brhint` instructions in Fig. 7.19. As

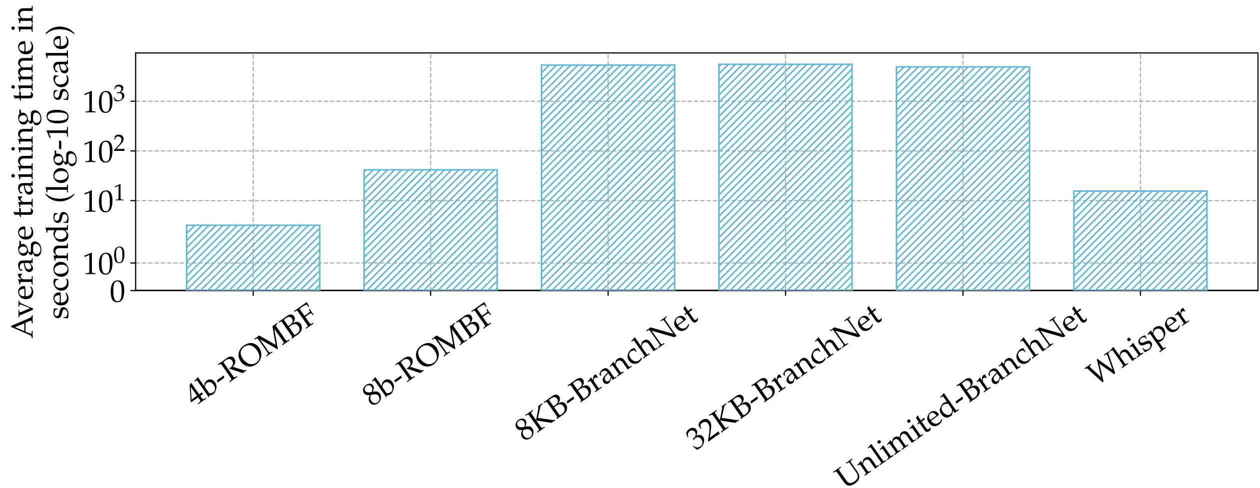


Figure 7.16: Average training time for Whisper compared to prior techniques (the y-axis is log-10 scale): BranchNet requires training times of more than thousands of seconds, even when trained on an NVIDIA Tesla V100 GPU. The training time for ROMBF grows exponentially with an increase in history length. The training time for Whisper is significantly lower than training times for 8-bit ROMBF and BranchNet.

Table 7.3: Different design parameters' values.

Design parameter	Value	Design parameter	Value
Minimum history length	8	Length of the hashed history	8
Maximum history length	1024	Logical operations used	4
Different history lengths	16	Hint buffer's size	32

shown, on average, Whisper increases these applications' static footprint by 11.4% (9.8%-13%) while introducing 9.8% (5.3%-14.7%) extra dynamic instructions.

Sensitivity analysis. As we describe in §7.3, Whisper's design includes several parameters including a minimum, maximum, and different history lengths, hashed history length, different logical operations used, and hint buffer's size. We determine these parameters' values empirically via sensitivity studies. For brevity, we do not present detailed results corresponding to these studies. As a summary, Table 7.3 shows these parameters' values we use to evaluate Whisper.

128KB TAGE-SC-L as baseline. We evaluate Whisper's effectiveness for a much larger, 128KB TAGE-SC-L baseline and show the results in Fig. 7.20. The 128KB TAGE-SC-L exhibits a branch-MPKI in the range of 0.4-5.4 (2.4 on average) for 12 data center applications. As shown, Whisper achieves an average misprediction reduction of 13.4% over the 128KB TAGE-SC-L baseline highlighting Whisper's effectiveness even for a larger TAGE-SC-L branch predictor.

Predictor size. We evaluate Whisper's sensitivity to the baseline branch predictor's size by varying

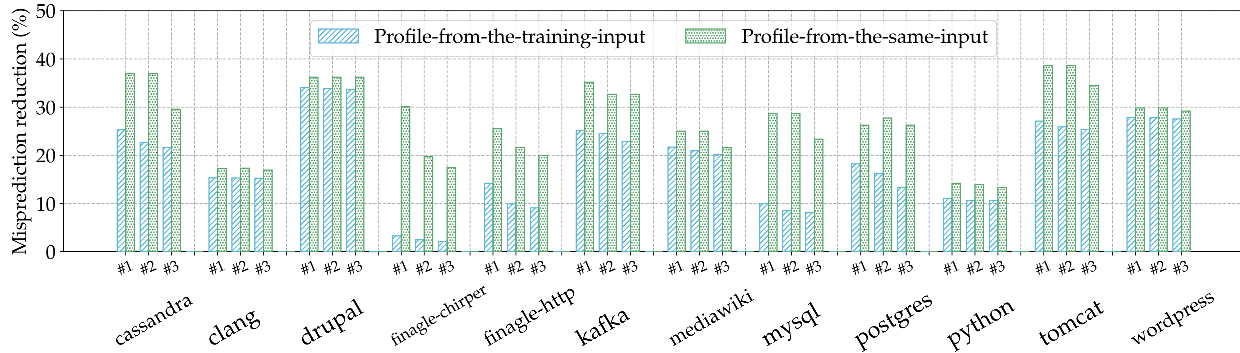


Figure 7.17: Whisper's performance for various application inputs: On average Whisper reduces 6.6% more branch mispredictions with input-specific profiles compared to profiles from different inputs.

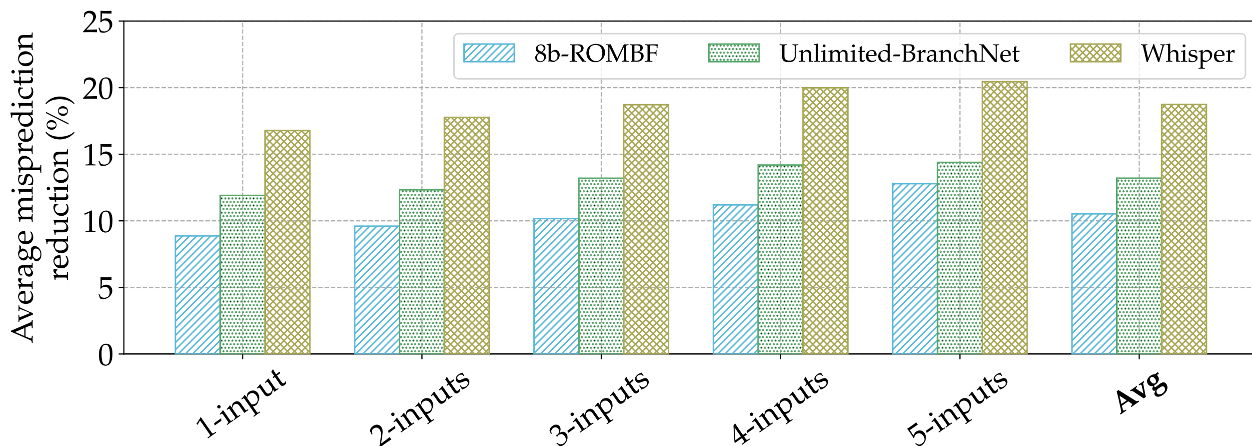


Figure 7.18: Whisper eliminates more branch mispredictions after merging profiles from various inputs.

TAGE-SC-L's capacity from 8KB to 1MB. Fig. 7.21 shows the results. As shown, Whisper consistently reduces more than 10% of all mispredictions irrespective of the predictor's capacity. Even the 1MB TAGE-SC-L incurs an average branch-MPKI of 1.9 compared to MTAGE-SC's branch-MPKI of 1.4. As even the 1MB TAGE-SC-L suffers from capacity and conflict mispredictions, Whisper still has the potential to reduce a significant number of mispredictions. Consequently, Whisper reduces mispredictions by 11.2% for the 1MB TAGE-SC-L.

Predictor warm-up. We evaluate Whisper's sensitivity to baseline branch predictor's (TAGE-SC-L) state by varying % of warm-up instructions from 0% to 90%. Fig. 7.22 shows the results. As shown, Whisper reduces all mispredictions TAGE-SC-L incurs by 17.5% without any warm-up. As TAGE-SC-L's warm-up period increases and TAGE-SC-L incurs fewer mispredictions, Whisper's average misprediction reduction (%) over to TAGE-SC-L drops slightly. Nevertheless, Whisper

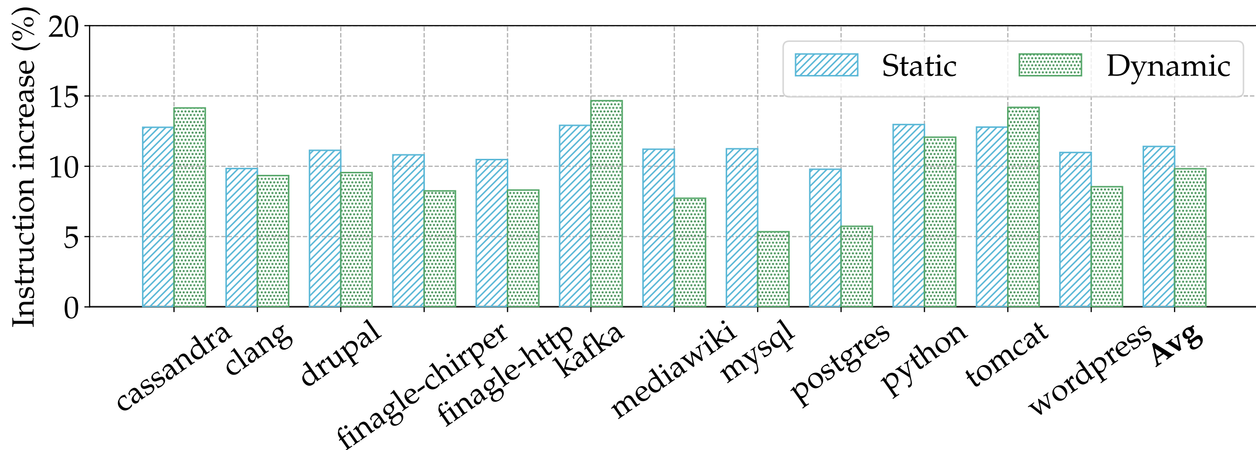


Figure 7.19: Whisper's overhead in static and dynamic instruction increase: on average, Whisper incurs a static overhead of 11.4% (9.8%-13%) and executes 9.8% (5.3%-14.7%) extra dynamic instructions due to `brhint` instructions.

still avoids a large number of mispredictions as it reduces TAGE-SC-L's mispredictions by 16.8% even when warm-up instructions account for 50% of all instructions.

Simulated instructions. We evaluate Whisper's sensitivity to the total number of instructions simulated by varying the number of instructions from 100 million to 1 billion. Fig. 7.23 shows the results. As shown, Whisper reduces 14.7% of all mispredictions even when one billion instructions are simulated.

7.6 Related Work

PGO for data center applications. The large instruction footprint and software complexity of modern data center applications make them a prime target for PGO [39, 41, 112, 179, 204, 344, 224]. Prior PGO techniques include code layout optimizations [65, 238, 136, 410, 286, 221, 236, 278, 279, 126, 139, 407], I-cache prefetching [41, 199] and replacement [202], and BTB prefetching [195] and replacement [334]. These techniques primarily focus on reducing frontend stalls while Whisper focuses on reducing branch mispredictions. Consequently, Whisper should be equally effective even in the presence of these techniques.

Online branch predictors. Most state-of-the-art online branch predictors are variants of TAGE [321] and Perceptron [167]. TAGE hashes global branch and path histories of different lengths to index into various tables composed of tagged saturating counters. TAGE-SC-L [317, 319], which won CBP-5 [18], is a popular TAGE variant that uses additional loop predictor and statistical corrector components to improve accuracy. Perceptron-based predictors, such as the Multiperspective Perceptron [162, 163], use a single-layer neural network to compute a sum of

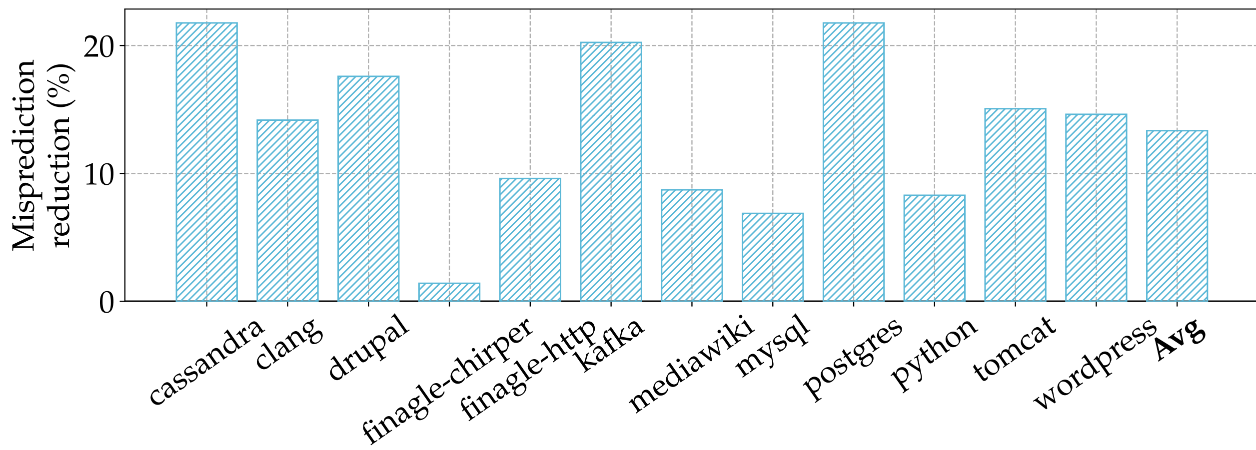


Figure 7.20: Whisper's reduction in branch mispredictions over the 128KB TAGE-SC-L baseline: Whisper reduces 13.4% of all mispredictions the 128KB TAGE-SC-L incurs.

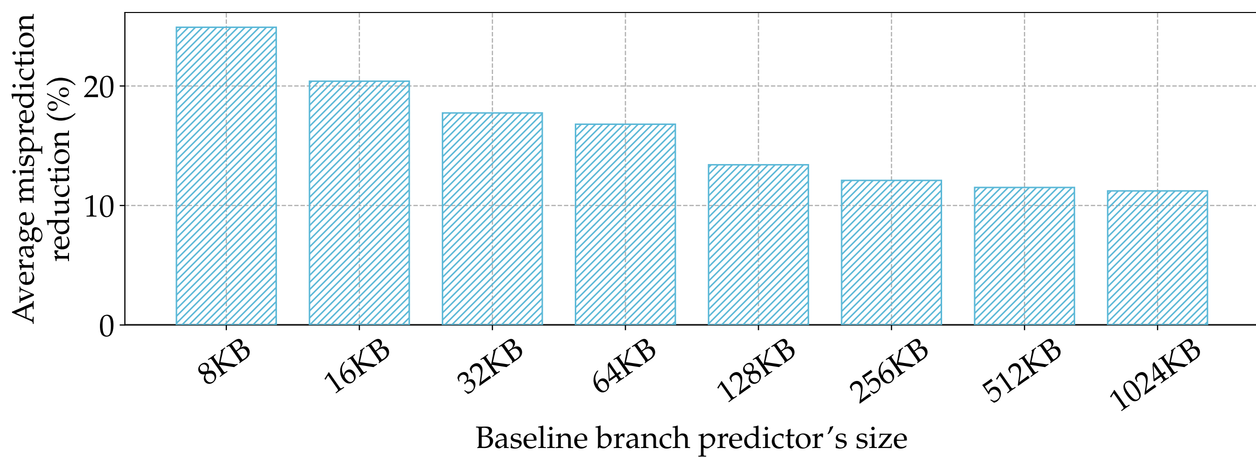


Figure 7.21: Whisper's performance for various baseline branch predictor's sizes: Whisper reduces even 1MB TAGE-SC-L's mispredictions by 11.2%.

weights that represent a learned correlation in branch history. A fundamental limitation of TAGE and Perceptron-based predictors is their inability to learn increasingly complex branch histories due to storage and run-time constraints. Other work in online branch prediction includes domain-specific branch predictors and predictors targeting data-dependent branches [296, 133, 309, 340].

Considering prior limitations, Whisper still leverages online branch predictors in the common case. Offline profiling and hardware support for ROMBF are then used to predict branches that online predictors struggle to predict accurately. This approach allows Whisper to reduce the resource burden placed on traditional online predictors from applications with noisy branch histories. Also, Whisper does not attempt to alter existing online branch predictors in hardware, which simplifies its implementation in modern processors.

Offline methods for branch prediction. Offline techniques, such as profiling and compiler-

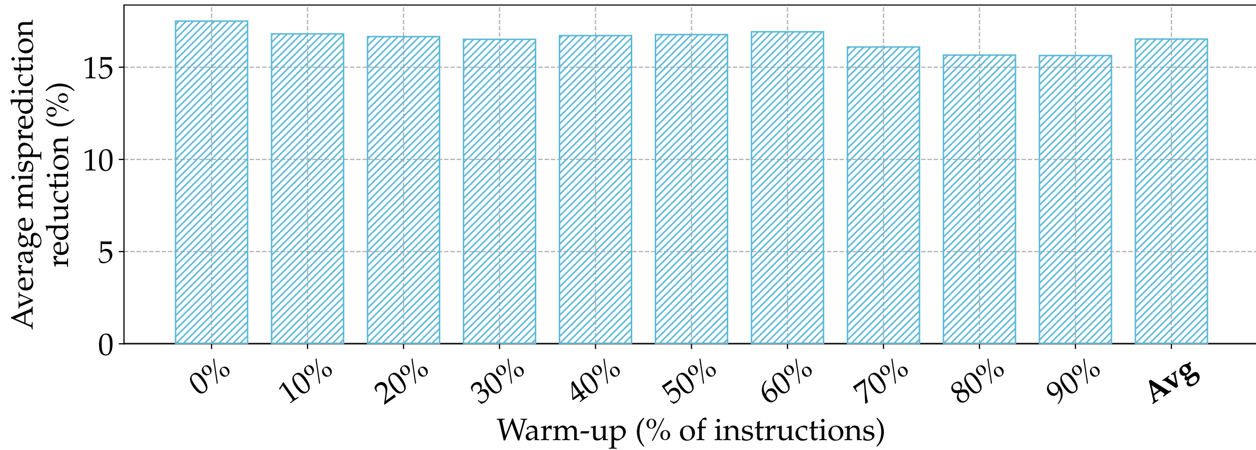


Figure 7.22: Whisper's performance for various TAGE-SC-L warm-up periods: Whisper reduces 16.8% of TAGE-SC-L's mispredictions with 50% of instructions considered as warm-up. On the other hand, Whisper avoids 17.5% of TAGE-SC-L's mispredictions without any warm-up.

based optimizations, have been used extensively to improve accuracy for branch prediction [115, 45, 392, 401, 210, 62, 282, 401, 164, 166, 160, 325, 356, 346, 111, 25, 363, 95, 243, 357]. BranchNet [405] is a recent offline method for reducing branch mispredictions. It uses CNNs, with a hardware-based inference component, to handle branches that online predictors struggle to predict accurately. The main limitation of BranchNet is its resource requirements (*i.e.*, multiple GPUs for efficient training, one CNN model per static branch) and implementation complexity in hardware. Whereas for Whisper, "training" or analyzing execution profiles can be done relatively cheaply using commodity CPUs and the hardware implementation is less demanding than hardware inference for deep learning. Additionally, BranchNet struggles to cover mispredictions spread out across many unique static branches. Whisper has less overhead per static branch due to the lightweight design of ROMBF compared to a CNN model in BranchNet.

7.7 Conclusion

The state-of-the-art branch predictor, TAGE-SC-L, suffers frequent branch mispredictions for data center applications as their large branch footprints overwhelm TAGE-SC-L's 64KB capacity. We propose, Whisper, a profile-guided hardware/software mechanism to efficiently reduce branch mispredictions in these data center applications through extended Read-Once Monotone Boolean Formulas that encode hard-to-predict correlations in branch history. Whisper inserts lightweight formulas in application code at link time using a new `brhint` instruction that is complemented by micro-architectural support for ROMBF. Through efficient offline analysis of application profiles, only select branches use these new micro-architectural changes; the remaining are predicted using

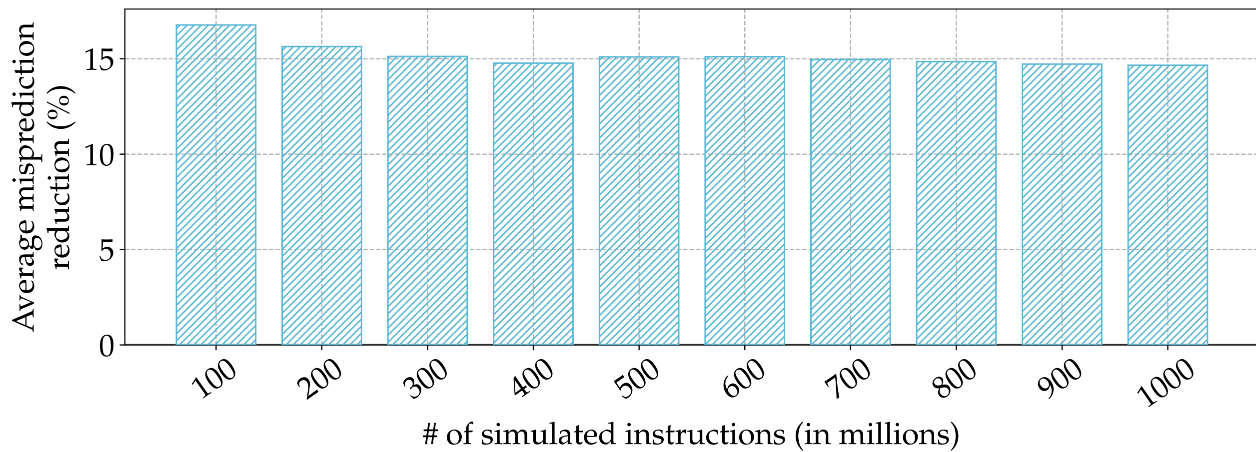


Figure 7.23: Whisper's performance for various numbers of simulated instructions: on average, Whisper avoids 14.7% of all mispredictions after simulating one billion instructions.

the underlying branch predictor – requiring no changes to the predictor itself. On average, Whisper reduces 16.8% (1.7%-32.4%) of branch mispredictions over TAGE-SC-L for 12 widely-used data center applications, with an average speedup of 2.8% (0.4%-4.6%), and outperforms existing profile-guided branch prediction mechanisms, such as BranchNet, by 7.9%.

CHAPTER 8

Conclusion and Future work: Democratizing hardware/software co-design

My research vision is to rethink computer systems from the ground up for efficiency. My work enabling efficient data center processing is one key step toward this vision. Going forward, I will democratize hardware/software co-design by building open-source systems and benchmarking methodologies. Using insights from leading these efforts, I will propose new techniques to solve a broad set of efficiency problems across the systems stack.

Democratizing systems research on production web services. Production web services widely deployed in today's data centers are difficult to study outside of the industry. As systematic investigations of these workloads are key to driving future innovation, I am currently working with Google researchers to democratize such investigations. Toward this goal, we have recently released the first version of Google's workload traces [6]. Next, I will build an open-source infrastructure that will enable end-to-end hardware/software co-design (*e.g.*, simulating clusters with networking) of these web services. To aid this effort, Google has awarded me a gift that I will use to launch my research lab.

Enabling hardware development to catch up to rapid software evolution. Software code base of data center applications evolves at a very rapid pace (*e.g.*, O(days)). In contrast, hardware evolution is significantly slower (*e.g.*, O(years)), making hardware design challenging as it is difficult for the hardware to keep up with new software trends and bottlenecks. To enable hardware to meet new software needs in a timely manner, I will design an end-to-end system that will enable hardware development to be guided by future software trends. This system will first profile an application's historical software evolution to identify how ensuing hardware bottlenecks evolve over time (*i.e.*, across application versions). Then, using this historical bottleneck evolution, the system will project how these key hardware bottlenecks might evolve in the future. Finally, the system will use this projection to apply automated compiler transformations that will generate a future application version exhibiting these projected bottlenecks. Showing interest, Intel has recently supported my initiative with a grant.

Managing thread oversubscription with hardware/software co-design. At any given time, even a single data center application have hundreds of threads ready to run on a single processor core, causing severe performance issues such as frequent context switching and interference among threads. Hardware/software co-design is a promising direction to address these problems. In the hardware theme, I will design techniques to quickly warm up micro-architectural structures, reducing the overhead of context switching. In the software theme, to reduce inference among threads, I will design mechanisms to run threads with similar access patterns (*e.g.*, to the micro-architectural structures) one after the other.

Hardware/software co-design for multi-tenant efficiency. Diverse data center applications (*e.g.*, latency-sensitive vs. batch-processing) are co-scheduled on the same machine to achieve higher resource utilization. However, this co-scheduling incurs a heavy cost as applications thrash each other out of micro-architectural structures. Using hardware/software co-design, I will propose efficient co-scheduling strategies that can avoid this destructive thrashing. In particular, I will design hardware techniques to identify different applications' demands for different micro-architectural structures. I will also propose software techniques that will use these demand vectors to schedule applications in a cooperative manner.

BIBLIOGRAPHY

[1] Adding processor trace support to linux. https://lwn.net/Articles/648154/.

[2] Apache cassandra. http://cassandra.apache.org/.

[3] Apache kafka. https://kafka.apache.org/powered-by.

[4] Apache tomcat. https://tomcat.apache.org/.

[5] Clang c language family frontend for llvm. [Online; accessed 19-Nov-2021].

[6] Google workload traces. https://dynamorio.org/google_workload_traces.html.

[7] An introduction to last branch records. https://lwn.net/Articles/680985/.

[8] Mysql. [Online; accessed 19-Nov-2021].

[9] Postgresql: Documentation: 14: pgbench. [Online; accessed 19-Nov-2021].

[10] Postgresql: The world's most advanced open source database. [Online; accessed 19-Nov-2021].

[11] Prefetchh: Prefetch data into caches (x86 instruction set reference). https://c9x.me/x86/html/file_module_x86_id_252.html. [Online; accessed 28-March-2020].

[12] Prfm (literal) — a64. http://shell-storm.org/armv8-a/ISA_v85A_A64_xml_00bet8_OPT/xhtml/prfm_reg.html. [Online; accessed 28-March-2020].

[13] The python performance benchmark suite. [Online; accessed 19-Nov-2021].

[14] Scarab. https://github.com/hpsresearchgroup/scarab.

[15] Twitter finagle. https://twitter.github.io/finagle/.

[16] Verilator. https://www.veripool.org/wiki/verilator.

[17] Welcome to python.org. [Online; accessed 19-Nov-2021].

[18] Championship branch prediction. https://jilp.org/cbp2016/, 2016.

[19] facebookarchive/oss-performance: Scripts for benchmarking various php implementations when running open source software. https://github.com/facebookarchive/oss-performance, 2019. (Online; last accessed 15-November-2019).

[20] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[21] Jaume Abella, Antonio González, Xavier Vera, and Michael FP O'Boyle. Iatac: a smart predictor to turn-off l2 cache lines. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2(1):55–77, 2005.

[22] Bilge Acun, Benjamin Lee, Fiodar Kazhamiaka, Kiwan Maeng, Udit Gupta, Manoj Chakkaravarthy, David Brooks, and Carole-Jean Wu. Carbon explorer: A holistic framework for designing carbon aware datacenters. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 118–132, 2023.

[23] Keith Adams, Jason Evans, Bertrand Maher, Guilherme Ottoni, Andrew Paroski, Brett Simmers, Edwin Smith, and Owen Yamauchi. The hiphop virtual machine. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 777–790, 2014.

[24] Narasimha Adiga, James Bonanno, Adam Collura, Matthias Heizmann, Brian R Prasky, and Anthony Saporito. The ibm z15 high frequency mainframe branch predictor industrial product. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 27–39. IEEE, 2020.

[25] Almutaz Adileh, David J. Lilja, and Lieven Eeckhout. Architectural support for probabilistic branches. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, page 108–120. IEEE Press, 2018.

[26] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[27] Sam Ainsworth and Timothy M. Jones. Software prefetching for indirect memory accesses. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, pages 305–317, Piscataway, NJ, USA, 2017. IEEE Press.

[28] Samira Mirbagher Ajorpaz, Elba Garza, Sangam Jindal, and Daniel A Jiménez. Exploring predictive replacement policies for instruction cache and branch target buffer. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 519–532. IEEE, 2018.

[29] Mohammad Mejbah Ul Alam, Tongping Liu, Guangming Zeng, and Abdullah Muzahid. Syncperf: Categorizing, detecting, and diagnosing synchronization performance bugs. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 298–313, 2017.

[30] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[31] DW Anderson, FJ Sparacio, and Robert M Tomasulo. The ibm system/360 model 91: Machine philosophy and instruction-handling. *IBM Journal of Research and Development*, 1967.

[32] Jennifer M Anderson and Monica S Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *ACM Sigplan Notices*, volume 28, pages 112–125. ACM, 1993.

[33] Ali Ansari, Fatemeh Golshan, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. Mana: Microarchitecting an instruction prefetcher. *The First Instruction Prefetching Championship*, 2020.

[34] Ali Ansari, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. Divide and conquer frontend bottleneck. In *Proceedings of the 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[35] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[36] Jon Petter Asen, Jo Inge Buskenes, Carl-Inge Colombo Nilsen, Andreas Austeng, and Sverre Holm. Implementing capon beamforming on a gpu for real-time cardiac ultrasound imaging. *IEEE transactions on ultrasonics, ferroelectrics, and frequency control*, 61(1):76–85, 2014.

[37] Truls Asheim, Boris Grot, and Rakesh Kumar. Btb-x: A storage-effective btb organization. *IEEE Computer Architecture Letters*, 2021.

[38] Andrew Ayers, Richard Schooler, and Robert Gottlieb. Aggressive inlining. *ACM SIGPLAN Notices*, 1997.

[39] Grant Ayers, Jung Ho Ahn, Christos Kozyrakis, and Parthasarathy Ranganathan. Memory hierarchy for web search. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 643–656. IEEE, 2018.

[40] Grant Ayers, Heiner Litz, Christos Kozyrakis, and Parthasarathy Ranganathan. Classifying memory access patterns for prefetching. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 513–526, 2020.

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

[42] Reza Azimi, Michael Stumm, and Robert W Wisniewski. Online performance analysis by statistical sampling of microprocessor performance counters. In *Proceedings of the 19th annual international conference on Supercomputing*, pages 101–110, 2005.

[43] David Bailey, Tim Harris, William Saphir, Rob Van Der Wijngaart, Alex Woo, and Maurice Yarrow. The nas parallel benchmarks 2.0. Technical report, Technical Report NAS-95-020, NASA Ames Research Center, 1995.

[44] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M Tamer Özsu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 362–373. IEEE, 2013.

[45] Thomas Ball and James R Larus. Branch prediction for free. *ACM SIGPLAN Notices*, 28(6):300–313, 1993.

[46] Prithviraj Banerjee, John A Chandy, Manish Gupta, Eugene W Hodges, John G Holm, Antonio Lain, Daniel J Palermo, Shankar Ramaswamy, and Ernesto Su. The paradigm compiler for distributed-memory multicomputers. *Computer*, 28(10):37–47, 1995.

[47] Sorav Bansal and Dharmendra S Modha. Car: Clock with adaptive replacement. In *FAST*, volume 4, pages 187–200, 2004.

[48] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2):78–101, 1966.

[49] Emery D Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: safe multithreaded programming for c/c++. In *ACM sigplan notices*, volume 44, pages 81–96. ACM, 2009.

[50] Christian Bienia, Sanjeev Kumar, and Kai Li. Parsec vs. splash-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 47–56. IEEE, 2008.

[51] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. The dacapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 169–190, 2006.

[52] William J Bolosky and Michael L Scott. False sharing and its effect on shared memory performance. In *Proceedings of the Fourth symposium on Experiences with distributed and multiprocessor systems*, 1993.

[53] James Bonanno, Adam Collura, Daniel Lipetz, Ulrich Mayer, Brian Prasky, and Anthony Saporito. Two level bulk preload branch prediction. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 71–82. IEEE, 2013.

[54] Michael D Bond and Kathryn S McKinley. Continuous path and edge profiling. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 130–140. IEEE Computer Society, 2005.

[55] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Acm Sigplan Notices*, volume 43, pages 101–113. ACM, 2008.

[56] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. An improved construction for counting bloom filters. In *European Symposium on Algorithms*, pages 684–695. Springer, 2006.

[57] Peter Braun and Heiner Litz. Understanding memory access patterns for prefetching. In *International Workshop on AI-assisted Design for Architecture (AIDArc), held in conjunction with ISCA*, 2019.

[58] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pages 265–275. IEEE, 2003.

[59] Jake Brutlag. Speed matters for google web search, 2009.

[60] Ioana Burcea and Andreas Moshovos. Phantom-btb: a virtualized branch target buffer design. *Acm Sigplan Notices*, 44(3):313–324, 2009.

[61] Michael Butler, Leslie Barnes, Debjit Das Sarma, and Bob Gelinas. Bulldozer: An approach to multithreaded compute performance. *IEEE Micro*, 31(2):6–15, 2011.

[62] Brad Calder, Dirk Grunwald, Michael Jones, Donald Lindsay, James Martin, Michael Mozer, and Benjamin Zorn. Evidence-based static branch prediction using machine learning. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(1):188–222, 1997.

[63] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. Compiler optimizations for improving data locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, pages 252–262, New York, NY, USA, 1994. ACM.

[64] Milind Chabbi, Shasha Wen, and Xu Liu. Featherlight on-the-fly false-sharing detection. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 152–167. ACM, 2018.

[65] Dehao Chen, David Xinliang Li, and Tipp Moseley. Autofdo: Automatic feedback-directed optimization for warehouse-scale applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 12–23. ACM, 2016.

[66] Dong Chen, Fangzhou Liu, Chen Ding, and Sreepathi Pai. Locality analysis through static parallel sampling. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 557–570, 2018.

[67] Jia Chen. Andersen's inclusion-based pointer analysis re-implementation in LLVM. https://github.com/grievejia/andersen, 2018. [Online; accessed 16-Nov-2018].

[68] Tien-Fu Chen and Jean-Loup Baer. Reducing memory latency via non-blocking and prefetching caches. *ACM SIGPLAN Notices*, 27(9):51–61, 1992.

[69] W. Y. Chen, P. P. Chang, T. M. Conte, and W. W. Hwu. The effect of code expanding optimizations on instruction cache design. *IEEE Trans. Comput.*, 42(9):1045–1057, September 1993.

[70] Wen-ke Chen, Sanjay Bhansali, Trishul Chilimbi, Xiaofeng Gao, and Weihaw Chuang. Profile-guided proactive garbage collection for locality optimization. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 332–340, New York, NY, USA, 2006. ACM.

[71] Mei-Ling Chiang, Chieh-Jui Yang, and Shu-Wei Tu. Kernel mechanisms with dynamic task-aware scheduling to reduce resource contention in numa multi-core systems. *Journal of Systems and Software*, 121:72–87, 2016.

[72] Trishul M Chilimbi, Bob Davidson, and James R Larus. Cache-conscious structure definition. In *ACM SIGPLAN Notices*, volume 34, pages 13–24. ACM, 1999.

[73] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 1–12, New York, NY, USA, 1999. ACM.

[74] Trishul M Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *OSDI*, volume 14, pages 571–582, 2014.

[75] Jyh-Herng Chow and Vivek Sarkar. False sharing elimination by selection of runtime scheduling parameters. In *Parallel Processing, 1997., Proceedings of the 1997 International Conference on*, pages 396–403. IEEE, 1997.

[76] Daniel Clifford, Hannes Payer, Michael Stanton, and Ben L Titzer. Memento mori: dynamic allocation-site-based optimizations. In *ACM SIGPLAN Notices*, volume 50, pages 105–117. ACM, 2015.

[77] cloudfare. kyotocabinet/kcstashtest.cc at master - cloudflare/kyotocabinet. https://github.com/cloudflare/kyotocabinet/blob/master/kcstashtest.cc, 2013. [Online; accessed 4-April-2019].

[78] Robert Cohn and P Geoffrey Lowney. Hot cold optimization of large windows/nt applications. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 80–89. IEEE, 1996.

[79] Alexander Collins, Tim Harris, Murray Cole, and Christian Fensch. Lira: Adaptive contention-aware thread placement for parallel runtime systems. In *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers*, page 2. ACM, 2015.

[80] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[81] Emilio Coppa, Camil Demetrescu, and Irene Finocchi. Input-sensitive profiling. *ACM SIGPLAN Notices*, 47(6):89–98, 2012.

[82] Emilio Coppa, Camil Demetrescu, Irene Finocchi, and Romolo Marotta. Multithreaded input-sensitive profiling. *arXiv preprint arXiv:1304.3804*, 2013.

[83] Intel Corparation. Intel (r) 64 and ia-32 architectures software developer's manual. *Combined Volumes, Dec*, 2016.

[84] Transaction Processing Performance Council. Tpc-c. [Online; accessed 19-Nov-2021].

[85] Transaction Processing Performance Council. Tpc-h. [Online; accessed 23-April-2019].

[86] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. {REPT}: Reverse debugging of failures in deployed software. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 17–32, 2018.

[87] Charlie Curtsinger and Emery D. Berger. Stabilizer: Statistically sound performance evaluation. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 219–228, New York, NY, USA, 2013. Association for Computing Machinery.

[88] Charlie Curtsinger and Emery D Berger. Coz: Finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 184–197, 2015.

[89] Daniel Lemire. Is software prefetching (__builtin_prefetch) useful for performance?, 2018. [Online; accessed 24-April-2019].

[90] Florian David, Gael Thomas, Julia Lawall, and Gilles Muller. Continuously measuring critical section pressure with the free-lunch profiler. *ACM SIGPLAN Notices*, 49(10):291–307, 2014.

[91] Florian David, Gaël Thomas, Julia L. Lawall, and Gilles Muller. Continuously measuring critical section pressure with the free-lunch profiler. In *OOPSLA*, 2014.

[92] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[93] Christian DeLozier, Ariel Eizenberg, Shiliang Hu, Gilles Pokam, and Joseph Devietti. Tmi: thread memory isolation for false sharing repair. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 639–650. ACM, 2017.

[94] Peter J Denning. Thrashing: Its causes and prevention. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, pages 915–922, 1968.

[95] Veerle Desmet, Lieven Eeckhout, and Koen De Bosschere. Using decision trees to improve program-based and profile-based static branch prediction. In *Asia-Pacific Conference on Advances in Computer Systems Architecture*, pages 336–352. Springer, 2005.

[96] David Dice. False sharing induced by card table marking. https://blogs.oracle.com/dave/false-sharing-induced-by-card-table-marking, 2011. [Online; last accessed 05-August-2018].

[97] Chen Ding and Yutao Zhong. Predicting whole-program locality through reuse distance analysis. In *Acm Sigplan Notices*, volume 38, pages 245–257. ACM, 2003.

[98] Wei Ding and Mahmut Kandemir. Capri: Cache-conscious data reordering for irregular codes. *ACM SIGMETRICS Performance Evaluation Review*, 42(1):477–489, 2014.

[99] Jack Doweck, Wen-Fu Kao, Allen Kuan-yu Lu, Julius Mandelblat, Anirudha Rahatekar, Lihu Rappoport, Efraim Rotem, Ahmad Yasin, and Adi Yoaz. Inside 6th-generation intel core: New microarchitecture code-named skylake. *IEEE Micro*, 2017.

[100] Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero, and Alexander V Veidenbaum. Improving cache management policies using dynamic reuse distances. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 389–400. IEEE, 2012.

[101] Richard Durstenfeld. Algorithm 235: random permutation. *Communications of the ACM*, 7(7):420, 1964.

[102] Tyler Dwyer and Alexandra Fedorova. On instruction organization. In *15th Workshop on Hot Topics in Operating Systems (HotOS {XV})*, 2015.

[103] efeslab. Huron: A false sharing detection and repair tool. https://github.com/efeslab/huron, 2019. [Online; accessed 12-April-2019].

[104] Ariel Eizenberg, Shiliang Hu, Gilles Pokam, and Joseph Devietti. Remix: online detection and repair of cache contention for the jvm. In *ACM SIGPLAN Notices*, volume 51, pages 251–265. ACM, 2016.

[105] Walter Erquinigo, David Carrillo-Cisneros, and Alston Tang. Reverse debugging at scale. https://engineering.fb.com/2021/04/27/developer-tools/reverse-debugging/.

[106] Facebook. Rocksdb: A persistent key-value store for flash and ram storage. https://github.com/facebook/rocksdb/, 2021.

[107] Barry Fagin. Partial resolution in branch target buffers. *IEEE Transactions on Computers*, 46(10):1142–1145, 1997.

[108] Priyank Faldu and Boris Grot. Leeway: Addressing variability in dead-block prediction for last-level caches. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 180–193. IEEE, 2017.

[109] Babak Falsafi and Thomas F Wenisch. A primer on hardware prefetching. *Synthesis Lectures on Computer Architecture*, 9(1):1–67, 2014.

[110] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM transactions on networking*, 8(3):281–293, 2000.

[111] M Umar Farooq, Lizy K John, et al. Store-load-branch (slb) predictor: A compiler assisted branch prediction for data dependent branches. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 59–70. IEEE, 2013.

[112] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *ACM SIGPLAN Notices*, volume 47, pages 37–48. ACM, 2012.

[113] Michael Ferdman, Cansu Kaynak, and Babak Falsafi. Proactive instruction fetch. In *International Symposium on Microarchitecture*, 2011.

[114] Michael Ferdman, Thomas F Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. Temporal instruction fetch streaming. In *International Symposium on Microarchitecture*, 2008.

[115] Joseph A Fisher and Stefan M Freudenberger. Predicting conditional branch directions from previous runs of a program. *ACM SIGPLAN Notices*, 27(9):85–95, 1992.

[116] Ronald Aylmer Fisher and Frank Yates. *Statistical tables for biological, agricultural and medical research*. Hafner Publishing Company, 1953.

[117] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 124, 2004.

[118] Hongliang Gao and Chris Wilkerson. A dueling segmented lru replacement algorithm with adaptive bypassing. 2010.

[119] Saturnino Garcia, Donghwan Jeon, Christopher M Louie, and Michael Bedford Taylor. Kremlin: rethinking and rebooting gprof for the multicore age. *ACM SIGPLAN Notices*, 46(6):458–469, 2011.

[120] Xinyang Ge, Weidong Cui, and Trent Jaeger. Griffin: Guarding control flows using intel processor trace. *ACM SIGPLAN Notices*, 2017.

[121] Xinyang Ge, Ben Niu, and Weidong Cui. Reverse debugging of kernel failures in deployed systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 281–292. USENIX Association, July 2020.

[122] Nathan Gober, Gino Chacon, Daniel Jiménez, and Paul V Gratz. The temporal ancestry prefetcher.

[123] Olga Golovanevsky, Alon Dayan, Ayal Zaks, and David Edelsohn. Trace-based data layout optimizations for multi-core processors. In *International Conference on High-Performance Embedded Architectures and Compilers*, pages 81–95. Springer, 2010.

[124] Abraham Gonzalez, Aasheesh Kolli, Samira Khan, Sihang Liu, Vidushi Dadu, Sagar Karandikar, Jichuan Chang, Krste Asanovic, and Parthasarathy Ranganathan. Profiling hyperscale big data processing. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–16, 2023.

[125] Daniel Goodman, Georgios Varisteas, and Tim Harris. Pandia: comprehensive contention-sensitive thread placement. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 254–269. ACM, 2017.

[126] Google. Propeller: Profile guided optimizing large scale llvm-based relinker. https://github.com/google/llvm-propeller, 2020.

[127] Susan L Graham, Peter B Kessler, and Marshall K Mckusick. Gprof: A call graph execution profiler. *ACM Sigplan Notices*, 17(6):120–126, 1982.

[128] Daniel A Jiménez Paul V Gratz and Gino Chacon Nathan Gober. Barca: Branch agnostic region searching algorithm.

[129] Brian Grayson, Jeff Rupley, Gerald Zuraski Zuraski, Eric Quinnell, Daniel A Jiménez, Tarun Nakra, Paul Kitchin, Ryan Hensley, Edward Brekelbaum, Vikas Sinha, et al. Evolution of the samsung exynos cpu microarchitecture. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 40–51. IEEE, 2020.

[130] Joseph L Greathouse, Zhiqiang Ma, Matthew I Frank, Ramesh Peri, and Todd Austin. Demand-driven software race detection using hardware performance counters. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 165–176. ACM, 2011.

[131] The PostgreSQL Global Development Group. Line number 3225. https://github.com/postgres/postgres/blob/master/src/backend/executor/nodeHash.c.

[132] Stephan M Günther and Josef Weidendorfer. Assessing cache false sharing effects by dynamic binary instrumentation. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, pages 26–33. ACM, 2009.

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

PACT '20, page 305–316, New York, NY, USA, 2020. Association for Computing Machinery.

[134] Vishal Gupta, Neelu Shivprakash Kalani, and Biswabandan Panda. Run-jump-run: Bouquet of instruction pointer jumpers for high performance instruction prefetching.

[135] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 290–299, 2007.

[136] Stavros Harizopoulos and Anastassia Ailamaki. Steps towards cache-resident transaction processing. In *International conference on Very large data bases*, 2004.

[137] Milad Hashemi, Kevin Swersky, Jamie A Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. Learning memory access patterns. *arXiv preprint arXiv:1803.02329*, 2018.

[138] Kim Hazelwood and James E Smith. Exploring code cache eviction granularities in dynamic optimization systems. In *International Symposium on Code Generation and Optimization*, 2004.

[139] Wenlei He, Julián Mestre, Sergey Pupyrev, Lei Wang, and Hongtao Yu. Profile inference revisited. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–24, 2022.

[140] Yuxiong He, Charles E Leiserson, and William M Leiserson. The cilkview scalability analyzer. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 145–156, 2010.

[141] Ravi Hegde. Optimizing application performance on intel core microarchitecture using hardware-implemented prefetchers. *Intel Software Network*, 2008. [Online; accessed 5-December-2020].

[142] Mark D Hill and Alan Jay Smith. Evaluating associativity in cpu caches. *IEEE Transactions on Computers*, 38(12):1612–1630, 1989.

[143] Changwan Hong, Wenlei Bao, Albert Cohen, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, J Ramanujam, and Ponnuswamy Sadayappan. Effective padding of multidimensional arrays to avoid cache conflict misses. In *ACM SIGPLAN Notices*, volume 51, pages 129–144. ACM, 2016.

[144] Zhigang Hu, Stefanos Kaxiras, and Margaret Martonosi. Timekeeping in the memory system: predicting and optimizing memory behavior. In *Proceedings 29th Annual International Symposium on Computer Architecture*, pages 209–220. IEEE, 2002.

[145] Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J Eliot B. Moss, Zhenlin Wang, and Perry Cheng. The garbage collection advantage: Improving program locality. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 69–80, New York, NY, USA, 2004. ACM.

[146] Intel. Tutorial: Identifying false sharing - c sample code, Oct 2017.

[147] Intel. Front-end bound. https://software.intel.com/content/www/us/en/develop/documentation/vtune-help/top/reference/cpu-metrics-reference/front-end-bound.html, 2021.

[148] Yasuo Ishii, Mary Inaba, and Kei Hiraki. Access map pattern matching for high performance data cache prefetch. *Journal of Instruction-Level Parallelism*, 13(2011):1–24, 2011.

[149] Yasuo Ishii, Mary Inaba, and Kei Hiraki. Unified memory optimizing architecture: memory subsystem control with a unified predictor. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 267–278, 2012.

[150] Yasuo Ishii, Jaekyu Lee, Krishnendra Nathella, and Dam Sunwoo. Rebasing instruction prefetching: An industry perspective. *IEEE Computer Architecture Letters*, 2020.

[151] Yasuo Ishii, Jaekyu Lee, Krishnendra Nathella, and Dam Sunwoo. Re-establishing fetch-directed instruction prefetching: An industry perspective. *IEEE International Symposium on Performance Analysis of Systems and Software*, 2021.

[152] Quinn Jacobson, Eric Rotenberg, and James E Smith. Path-based next trace prediction. In *Proceedings of 30th Annual International Symposium on Microarchitecture*, pages 14–23. IEEE, 1997.

[153] Akanksha Jain and Calvin Lin. Back to the future: leveraging belady's algorithm for improved cache replacement. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 78–89. IEEE, 2016.

[154] Akanksha Jain and Calvin Lin. Rethinking belady's algorithm to accommodate prefetching. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 110–123. IEEE, 2018.

[155] Akanksha Jain and Calvin Lin. Cache replacement policies. *Synthesis Lectures on Computer Architecture*, 14(1):1–87, 2019.

[156] Aamer Jaleel, Kevin B Theobald, Simon C Steely Jr, and Joel Emer. High performance cache replacement using re-reference interval prediction (rrip). *ACM SIGARCH Computer Architecture News*, 38(3):60–71, 2010.

[157] Saba Jamilan, Tanvir Ahmed Khan, Grant Ayers, Baris Kasikci, and Heiner Litz. Apt-get: Profile-guided timely software prefetching. EuroSys 2022.

[158] Sanath Jayasena, Saman Amarasinghe, Asanka Abeyweera, Gayashan Amarasinghe, Himeshi De Silva, Sunimal Rathnayake, Xiaoqiao Meng, and Yanbin Liu. Detection of false sharing using machine learning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 30. ACM, 2013.

[159] Tor E Jeremiassen and Susan J Eggers. *Reducing false sharing on shared memory multiprocessors through compile time data transformations*, volume 30. ACM, 1995.

[160] Daniel A Jiménez. Code placement for improving dynamic branch prediction accuracy. In *ACM SIGPLAN Notices*, volume 40, pages 107–116. ACM, 2005.

[161] Daniel A Jiménez. Insertion and promotion for tree-based pseudolru last-level caches. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 284–296, 2013.

[162] Daniel A. Jiménez. Multiperspective perceptron predictor. In *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, Seoul, South Korea, June 2016.

[163] Daniel A. Jiménez. Multiperspective perceptron predictor with tage. In *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, Seoul, South Korea, June 2016.

[164] Daniel A Jiménez, Heather L Hanson, and Calvin Lin. Boolean formula-based branch prediction for future technologies. In *PACT*. IEEE, 2001.

[165] Daniel A Jiménez, Stephen W Keckler, and Calvin Lin. The impact of delay on the design of branch predictors. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 67–76, 2000.

[166] Daniel A Jiménez and Calvin Lin. Branch path re-aliasing. In *Proceedings of the 4th Workshop on Feedback Directed and Dynamic Optimization (FDDO-4)*. Citeseer, 2001.

[167] Daniel A. Jiménez and Calvin Lin. Dynamic branch prediction with perceptrons. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture (HPCA'01), Nuevo Leone, Mexico, January 20-24, 2001*, pages 197–206. IEEE Computer Society, 2001.

[168] Daniel A Jiménez and Elvira Teran. Multiperspective reuse prediction. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 436–448. IEEE, 2017.

[169] José A Joao, M Aater Suleman, Onur Mutlu, and Yale N Patt. Bottleneck identification and scheduling in multithreaded applications. *ACM SIGARCH Computer Architecture News*, 40(1):223–234, 2012.

[170] jvm-profiling-tools. perf-map-agent, 2018. [Online; accessed 6-December-2020].

[171] Ismail Kadayif and Mahmut Kandemir. Quasidynamic layout optimizations for improving data locality. *IEEE Transactions on Parallel & Distributed Systems*, (11):996–1011, 2004.

[172] Melanie Kambadur, Kui Tang, and Martha A Kim. Harmony: Collection and analysis of parallel block vectors. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 452–463. IEEE, 2012.

[173] M Kandemir, A Choudhary, and J Ramanujam. Improving locality in out-of-core computations using data layout transformations. In *International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 359–366. Springer, 1998.

[174] Mahmut Kandemir, Alok Choudhary, J Ramanujam, and Prithviraj Banerjee. A framework for interprocedural locality optimization using both loop and data layout transformations. In *Parallel Processing, 1999. Proceedings. 1999 International Conference on*, pages 95–102. IEEE, 1999.

[175] Mahmut Kandemir, Alok Choudhary, J Ramanujam, and Prithviraj Banerjee. A graph based framework to detect optimal memory layouts for improving data locality. In *Parallel Processing, 1999. 13th International and 10th Symposium on Parallel and Distributed Processing, 1999. 1999 IPPS/SPDP. Proceedings*, pages 738–743. IEEE, 1999.

[176] Mahmut Kandemir, Alok Choudhary, Jagannathan Ramanujam, Nagaraj Shenoy, and Prithviraj Banerjee. Enhancing spatial locality via data layout optimizations. In *European Conference on Parallel Processing*, pages 422–434. Springer, 1998.

[177] Mahmut Kandemir, Alok Choudhary, J Ramaujam, and Prithviraj Banerjee. On reducing false sharing while improving locality on shared memory multiprocessors. In *Parallel Architectures and Compilation Techniques, 1999. Proceedings. 1999 International Conference on*, pages 203–211. IEEE, 1999.

[178] Mahmut Kandemir and Ismail Kadayif. Compiler-directed selection of dynamic memory layouts. In *Proceedings of the ninth international symposium on Hardware/software codesign*, pages 219–224. ACM, 2001.

[179] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 158–169, 2015.

[180] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 158–169, New York, NY, USA, 2015. ACM.

[181] Sagar Karandikar, Aniruddha N Udipi, Junsun Choi, Joonho Whangbo, Jerry Zhao, Svilen Kanev, Edwin Lim, Jyrki Alakuijala, Vrishab Madduri, Yakun Sophia Shao, et al. Cdpu: Co-designing compression and decompression processing units for hyperscale systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–17, 2023.

[182] Ramakrishna Karedla, J Spencer Love, and Bradley G Wherry. Caching strategies to improve disk system performance. *Computer*, 27(3):38–46, 1994.

[183] Baris Kasikci, Thomas Ball, George Candea, John Erickson, and Madanlal Musuvathi. Efficient tracing of cold code via bias-free sampling. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 243–254, 2014.

[184] Baris Kasikci, Weidong Cui, Xinyang Ge, and Ben Niu. Lazy diagnosis of in-production concurrency bugs. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 582–598, 2017.

[185] Baris Kasikci, Cristiano Pereira, Gilles Pokam, Benjamin Schubert, Madanlal Musuvathi, and George Candea. Failure sketches: A better way to debug. Hot Topics in Operating Systems, page 5, 2015.

[186] Baris Kasikci, Cristiano Pereira, Gilles Pokam, Benjamin Schubert, Malandal Musuvathi, and George Candea. Failure sketches: A better way to debug. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, May 2015. USENIX Association.

[187] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. Failure sketching: A technique for automated root cause diagnosis of in-production failures. In *SOSP*, Monterey, CA, October 2015.

[188] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. Failure sketching: A technique for automated root cause diagnosis of in-production failures. In *Proceedings of the 25th Symposium on Operating Systems Principles*, page 344–360, 2015.

[189] Cansu Kaynak, Boris Grot, and Babak Falsafi. Shift: Shared history instruction fetch for lean-core server processors. In *International Symposium on Microarchitecture*, 2013.

[190] Cansu Kaynak, Boris Grot, and Babak Falsafi. Confluence: unified instruction supply for scale-out servers. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 166–177, 2015.

[191] Ken Kennedy and Ulrich Kremer. Automatic data layout for high performance fortran. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, page 76. ACM, 1995.

[192] Muneeb Khan, Andreas Sandberg, and Erik Hagersten. A case for resource efficient prefetching in multicores. In *2014 43rd International Conference on Parallel Processing*, pages 101–110. IEEE, 2014.

[193] Samira Manabi Khan, Yingying Tian, and Daniel A Jimenez. Sampling dead block prediction for last-level caches. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 175–186. IEEE, 2010.

[194] Tanvir Ahmed Khan, Nathan Brown, Akshitha Sriraman, Niranjan Soundararajan, Rakesh Kumar, Joseph Devietti, Sreenivas Subramoney, Gilles Pokam, Heiner Litz, and Baris Kasikci. Twig: Profile-guided btb prefetching for data center applications. MICRO 2021.

[195] Tanvir Ahmed Khan, Nathan Brown, Akshitha Sriraman, Niranjan K Soundararajan, Rakesh Kumar, Joseph Devietti, Sreenivas Subramoney, Gilles A Pokam, Heiner Litz, and Baris Kasikci. Twig: Profile-guided btb prefetching for data center applications. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 816–829, 2021.

[196] Tanvir Ahmed Khan, Ian Neal, Gilles Pokam, Barzan Mozafari, and Baris Kasikci. Dmon: Efficient detection and correction of data locality problems using selective profiling. OSDI 2021.

[197] Tanvir Ahmed Khan, Ian Neal, Gilles Pokam, Barzan Mozafari, and Baris Kasikci. Dmon: Efficient detection and correction of data locality problems using selective profiling. In *Proceedings (to appear) of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, OSDI 2021. USENIX Association, July 2021.

[198] Tanvir Ahmed Khan, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. I-spy: Context-driven conditional instruction prefetching with coalescing. MICRO 2020.

[199] Tanvir Ahmed Khan, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. I-spy: Context-driven conditional instruction prefetching with coalescing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 146–159. IEEE, 2020.

[200] Tanvir Ahmed Khan, Muhammed Ugur, Krishnendra Nathella, Dam Sunwoo, Heiner Litz, Daniel A Jiménez, and Baris Kasikci. Whisper: Profile-guided branch misprediction elimination for data center applications. MICRO 2022.

[201] Tanvir Ahmed Khan, Dexin Zhang, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. Ripple: Profile-guided instruction cache replacement for data center applications. ISCA 2021.

[202] Tanvir Ahmed Khan, Dexin Zhang, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. Ripple: Profile-guided instruction cache replacement for data center applications. In *Proceedings of the 48th International Symposium on Computer Architecture (ISCA)*, ISCA 2021, June 2021.

[203] Tanvir Ahmed Khan, Yifan Zhao, Gilles Pokam, Barzan Mozafari, and Baris Kasikci. Huron : Hybrid false sharing detection and repair. PLDI 2019.

[204] Tanvir Ahmed Khan, Yifan Zhao, Gilles Pokam, Barzan Mozafari, and Baris Kasikci. Huron: hybrid false sharing detection and repair. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 453–468, 2019.

[205] Mazen Kharbutli and Yan Solihin. Counter-based cache replacement algorithms. In *2005 International Conference on Computer Design*, pages 61–68. IEEE, 2005.

[206] Jinchun Kim, Elvira Teran, Paul V Gratz, Daniel A Jiménez, Seth H Pugsley, and Chris Wilkerson. Kill the program counter: Reconstructing program behavior in the processor cache hierarchy. *ACM SIGPLAN Notices*, 52(4):737–749, 2017.

[207] Andi Kleen and Beeman Strong. Intel processor trace on linux. *Tracing Summit*, 2015.

[208] Ryotaro Kobayashi, Yuji Yamada, Hideki Ando, and Toshio Shimada. A cost-effective branch target buffer with a two-level table organization. In *Proceedings of the 2nd International Symposium of Low-Power and High-Speed Chips (COOL Chips II)*, 1999.

[209] Aasheesh Kolli, Ali Saidi, and Thomas F Wenisch. Rdip: return-address-stack directed instruction prefetching. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 260–271. IEEE, 2013.

[210] Andreas Krall. Improving semi-static branch prediction by code replication. *ACM SIGPLAN Notices*, 29(6):97–106, 1994.

[211] Rakesh Kumar, Boris Grot, and Vijay Nagarajan. Blasting through the front-end bottleneck with shotgun. *ACM SIGPLAN Notices*, 53(2):30–42, 2018.

[212] Rakesh Kumar, Cheng-Chieh Huang, Boris Grot, and Vijay Nagarajan. Boomerang: A metadata-free architecture for control flow delivery. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 493–504. IEEE, 2017.

[213] Randall L. Hyde and Brett D. Fleisch. An analysis of degenerate sharing and false coherence. 34:183–195, 05 1996.

[214] William Landi and Barbara G Ryder. Pointer-induced aliasing: A problem classification. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 93–103, 1991.

[215] Chris Lattner. Llvm and clang: Next generation compiler technology. In *The BSD conference*, pages 1–2, 2008.

[216] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[217] Rahman Lavaee, John Criswell, and Chen Ding. Codestitcher: inter-procedural basic block layout optimization. In *Proceedings of the 28th International Conference on Compiler Construction*, pages 65–75, 2019.

[218] Lee and Smith. Branch prediction strategies and branch target buffer design. *Computer*, 17(1):6–22, 1984.

[219] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. On the existence of a spectrum of policies that subsumes the least recently used (lru) and least frequently used (lfu) policies. In *Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 134–143, 1999.

[220] John Levon and Philippe Elie. Oprofile: A system profiler for linux, 2004.

[221] David Xinliang Li, Raksit Ashok, and Robert Hundt. Lightweight feedback-directed cross-module optimization. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 53–61. ACM, 2010.

[222] Ming Li, Shucheng Yu, Yao Zheng, Kui Ren, and Wenjing Lou. Scalable and secure sharing of personal health records in cloud computing using attribute-based encryption. *IEEE transactions on parallel and distributed systems*, 24(1):131–143, 2013.

[223] Jonathan Lifflander and Sriram Krishnamoorthy. Cache locality optimization for recursive programs. In *ACM SIGPLAN Notices*, volume 52, pages 1–16. ACM, 2017.

[224] Heiner Litz, Grant Ayers, and Parthasarathy Ranganathan. Crisp: critical slice prefetching. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 300–313, 2022.

[225] CL Liu. False sharing analysis for multithreaded programs. *Master's thesis, National Chung Cheng University*, 2009.

[226] Evan Zheran Liu, Milad Hashemi, Kevin Swersky, Parthasarathy Ranganathan, and Junwhan Ahn. An imitation learning approach for cache replacement. *arXiv preprint arXiv:2006.16239*, 2020.

[227] Haiming Liu, Michael Ferdman, Jaehyuk Huh, and Doug Burger. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 222–233. IEEE, 2008.

[228] Tongping Liu and Emery D Berger. Sheriff: precise detection and automatic mitigation of false sharing. *ACM Sigplan Notices*, 46(10):3–18, 2011.

[229] Tongping Liu and Xu Liu. Cheetah: detecting false sharing efficiently and effectively. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 1–11. ACM, 2016.

[230] Tongping Liu, Chen Tian, Ziang Hu, and Emery D. Berger. Predator: Predictive false sharing detection. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 3–14, New York, NY, USA, 2014. ACM.

[231] Tongping Liu, Guangming Zeng, Abdullah Muzahid, et al. Syncperf: Categorizing, detecting, and diagnosing synchronization performance bugs. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 298–313. ACM, 2017.

[232] Xu Liu and John Mellor-Crummey. Pinpointing data locality problems using data-centric analysis. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 171–180. IEEE Computer Society, 2011.

[233] Xu Liu and John Mellor-Crummey. A data-centric profiler for parallel programs. In *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2013.

[234] Xu Liu, Kamal Sharma, and John Mellor-Crummey. Arraytool: a lightweight profiler to guide array regrouping. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 405–415. IEEE, 2014.

[235] Xu Liu and Bo Wu. Scaanalyzer: A tool to identify memory scalability bottlenecks in parallel programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 47:1–47:12, New York, NY, USA, 2015. ACM.

[236] C-K Luk, Robert Muth, Harish Patil, Robert Cohn, and Geoff Lowney. Ispike: a post-link optimizer for the intel/spl reg/itanium/spl reg/architecture. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 15–26. IEEE, 2004.

[237] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40, pages 190–200. ACM, 2005.

[238] Chi-Keung Luk and Todd C Mowry. Cooperative prefetching: Compiler and hardware support for effective instruction prefetching in modern processors. In *International Symposium on Microarchitecture*, 1998.

[239] Liang Luo, Akshitha Sriraman, Brooke Fugate, Shiliang Hu, Gilles Pokam, Chris J Newburn, and Joseph Devietti. Laser: Light, accurate sharing detection and repair. In *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*, pages 261–273. IEEE, 2016.

[240] Piotr R Luszczek, David H Bailey, Jack J Dongarra, Jeremy Kepner, Robert F Lucas, Rolf Rabenseifner, and Daisuke Takahashi. The hpc challenge (hpcc) benchmark suite. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, volume 213. Citeseer, 2006.

[241] Peter S Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[242] Stanislav Manilov, Christos Vasiladiotis, and Björn Franke. Generalized profile-guided iterator recognition. In *Proceedings of the 27th International Conference on Compiler Construction*, pages 185–195, 2018.

[243] Yonghua Mao, Junjie Shen, and Xiaolin Gui. A study on deep belief net for branch prediction. *IEEE Access*, 6:10779–10786, 2017.

[244] Joe Mario. C2C - False Sharing Detection in Linux Perf. `https://joemario.github.io/blog/2016/09/01/c2c-blog/`, 2016. [Online; last accessed 04-August-2018].

[245] Markus Weninger. What exactly does -xx:-tieredcompilation do?, 2016. [Online; accessed 11-November-2019].

[246] Jason Mars and Lingjia Tang. Understanding application contentiousness and sensitivity on modern multicores. In *Advances in Computers*, volume 91, pages 59–85. Elsevier, 2013.

[247] Robert Martin, John Demme, and Simha Sethumadhavan. Timewarp: rethinking time-keeping and performance monitoring mechanisms to mitigate side-channel attacks. *ACM SIGARCH Computer Architecture News*, 40(3):118–129, 2012.

[248] Eric Masanet, Arman Shehabi, Nuoa Lei, Sarah Smith, and Jonathan Koomey. Recalibrating global data center energy-use estimates. *Science*, 367(6481):984–986, 2020.

[249] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems journal*, 9(2):78–117, 1970.

[250] Scott McFarling. Combining branch predictors. Technical report, Citeseer, 1993.

[251] Kathryn S. McKinley and Olivier Temam. A quantitative analysis of loop nest locality. In *ASPLOS-VII Proceedings - Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, Massachusetts, USA, October 1-5, 1996.*, pages 94–104, 1996.

[252] mcmcc. False sharing in boost::detail::spinlock_pool? https://stackoverflow.com/questions/11037655/false-sharing-in-boostdetailspinlock-pool, 2012. [Online; accessed 09-June-2018].

[253] Prashanth Menon, Todd C Mowry, and Andrew Pavlo. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *Proceedings of the VLDB Endowment*, 11(1):1–13, 2017.

[254] Pierre Michaud. Pips: Prefetching instructions with probabilistic scouts. In *The 1st Instruction Prefetching Championship*, 2020.

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

[256] Mircea Trofin. Support for cache prefetching profiles. by mtrofin · pull request #75 · google/autofdo, 2018. [Online; accessed 17-November-2019].

[257] Svetozar Miucin, Conor Brady, and Alexandra Fedorova. End-to-end memory behavior profiling with dinamite. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 1042–1046. ACM, 2016.

[258] Svetozar Miucin and Alexandra Fedorova. Data-driven spatial locality. In *Proceedings of the International Symposium on Memory Systems*, pages 243–253. ACM, 2018.

[259] Angélica Aparecida Moreira, Guilherme Ottoni, and Fernando Magno Quintão Pereira. Vespa: static profiling for binary optimization. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–28, 2021.

[260] Todd C Mowry. *Tolerating latency through software-controlled data prefetching*. PhD thesis, to the Department of Electrical Engineering.Stanford University, 1994.

[261] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS V, pages 62–73, New York, NY, USA, 1992. ACM.

[262] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sanchez. Exploiting locality in graph analytics through hardware-accelerated traversal scheduling. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–14. IEEE, 2018.

[263] Jumana Mundichipparakkal, Krishnendra Nathella, and Tanvir Ahmed Khan. Arm neoverse n1 core: Performance analysis methodology. https://armkeil.blob.core.windows.net/developer/Files/pdf/white-paper/neoverse-n1-core-performance-v2.pdf, 2021.

[264] Joseph Musmanno. Data intensive systems (dis) benchmark performance summary. Technical report, TITAN SYSTEMS CORP WALTHAM MA, 2003.

[265] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N Patt. Runahead execution: An effective alternative to large instruction windows. *IEEE Micro*, 2003.

[266] Nayana Prasad Nagendra, Grant Ayers, David I August, Hyoun Kyu Cho, Svilen Kanev, Christos Kozyrakis, Trivikram Krishnamurthy, Heiner Litz, Tipp Moseley, and Parthasarathy Ranganathan. Asmdb: Understanding and mitigating front-end stalls in warehouse-scale computers. *IEEE Micro*, 40(3):56–63, 2020.

[267] Ravi Nair. Dynamic path-based branch correlation. In *Proceedings of the 28th annual international symposium on Microarchitecture*, pages 15–23. IEEE, 1995.

[268] Tomoki Nakamura, Toru Koizumi, Yuya Degawa, Hidetsugu Irie, Shuichi Sakai, and Ryota Shioya. D-jolt: Distant jolt prefetcher.

[269] Mihir Nanavati, Mark Spear, Nathan Taylor, Shriram Rajagopalan, Dutch T Meyer, William Aiello, and Andrew Warfield. Whose cache line is it anyway?: operating system support for live detection and repair of false sharing. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 141–154. ACM, 2013.

[270] Ian Neal, Gefei Zuo, Eric Shiple, Tanvir Ahmed Khan, Youngjin Kwon, Simon Peter, and Baris Kasikci. Rethinking file mapping for persistent memory. FAST 2021.

[271] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[272] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 562–571. IEEE Press, 2013.

[273] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. *Acm Sigmod Record*, 22(2):297–306, 1993.

[274] Guilherme Ottoni. Hhvm jit: A profile-guided, region-based compiler for php and hack. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 151–165. ACM, 2018.

[275] Guilherme Ottoni and Bin Liu. Hhvm jump-start: Boosting both warmup and steady-state performance at scale. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 340–350. IEEE.

[276] Guilherme Ottoni and Bertrand Maher. Optimizing function placement for large-scale data-center applications. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, pages 233–244. IEEE Press, 2017.

[277] Shien-Tai Pan, Kimming So, and Joseph T Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. In *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, pages 76–84, 1992.

[278] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. Bolt: a practical binary optimizer for data centers and beyond. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, pages 2–14. IEEE Press, 2019.

[279] Maksim Panchenko, Rafael Auler, Laith Sakka, and Guilherme Ottoni. Lightning bolt: powerful, fast, and scalable binary optimization. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, pages 119–130, 2021.

[280] Reena Panda, Paul V Gratz, and Daniel A Jiménez. B-fetch: Branch prediction directed prefetching for in-order processors. *IEEE Computer Architecture Letters*, 11(2):41–44, 2011.

[281] Paratools. Threadspotter. http://threadspotter.paratools.com/, 2019. [Online; accessed 22-Oct-2019].

[282] Jason RC Patterson. Accurate static branch prediction by value range propagation. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 67–78, 1995.

[283] Andrea Pellegrini, Nigel Stephens, Magnus Bruce, Yasuo Ishii, Joseph Pusdesris, Abhishek Raja, Chris Abernathy, Jinson Koppanalil, Tushar Ringe, Ashok Tummala, et al. The arm neoverse n1 platform: Building blocks for the next-gen cloud-to-edge infrastructure soc. *IEEE Micro*, 40(2):53–62, 2020.

[284] Chris H Perleberg and Alan Jay Smith. Branch target buffer design and optimization. *IEEE transactions on computers*, 42(4):396–412, 1993.

[285] Aleksey Pesterev, Nickolai Zeldovich, and Robert T Morris. Locating cache performance bottlenecks using data profiling. In *Proceedings of the 5th European conference on Computer systems*, pages 335–348. ACM, 2010.

[286] Larry L Peterson. Architectural and compiler support for effective instruction prefetching: a cooperative approach. *ACM Transactions on Computer Systems*, 2001.

[287] Erez Petrank and Dror Rawitz. The hardness of cache conscious data placement. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, page 101–112, New York, NY, USA, 2002. Association for Computing Machinery.

[288] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 16–27, New York, NY, USA, 1990. ACM.

[289] Karl Pettis and Robert C Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 16–27, 1990.

[290] Jim Pierce and Trevor Mudge. Wrong-path instruction prefetching. In *International Symposium on Microarchitecture*, 1996.

[291] Orestis Polychroniou, Arun Raghavan, and Kenneth A Ross. Rethinking simd vectorization for in-memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1493–1508. ACM, 2015.

[292] PostgreSQL. Postgresql: The world's most advanced open source relational database. [Online; accessed 23-April-2019].

[293] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. Iterative optimization in the polyhedral model: Part II, multidimensional time. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, pages 90–100, Tucson, Arizona, June 2008. ACM Press.

[294] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. Renaissance: Benchmarking suite for parallel applications on the jvm. In *Programming Language Design and Implementation*, 2019.

[295] Aleksandar Prokopec, Andrea Rosa, David Leopoldseder, Gilles Duboscq, Petr Tuma, Martin Studener, Lubomir Bulej, Yudi Zheng, Alex Villazon, Doug Simon, et al. Renaissance: benchmarking suite for parallel applications on the jvm. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 31–47. ACM, 2019.

[296] Stephen Pruett and Yale 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, page 804–815, New York, NY, USA, 2021. Association for Computing Machinery.

[297] Moinuddin K Qureshi, Aamer Jaleel, Yale N Patt, Simon C Steely, and Joel Emer. Adaptive insertion policies for high performance caching. *ACM SIGARCH Computer Architecture News*, 35(2):381–391, 2007.

[298] Moinuddin K Qureshi, Daniel N Lynch, Onur Mutlu, and Yale N Patt. A case for mlp-aware cache replacement. In *33rd International Symposium on Computer Architecture (ISCA'06)*, pages 167–178. IEEE, 2006.

[299] Alex Ramirez, Luiz André Barroso, Kourosh Gharachorloo, Robert Cohn, Josep Larriba-Pey, P Geoffrey Lowney, and Mateo Valero. Code layout optimizations for transaction processing workloads. *ACM SIGARCH Computer Architecture News*, 2001.

[300] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 13–24. Ieee, 2007.

[301] Glenn Reinman, Todd Austin, and Brad Calder. A scalable front-end architecture for fast instruction delivery. *ACM SIGARCH Computer Architecture News*, 27(2):234–245, 1999.

[302] Glenn Reinman, Brad Calder, and Todd Austin. Fetch directed instruction prefetching. In *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 16–27. IEEE, 1999.

[303] Manman Ren and Shane Nay. Improving iOS Startup Performance with Binary Layout Optimizations, 2019. [Online; accessed 25-Oct-2019].

[304] Roman Oderov. Sampling and vtune's disadvantages, 2012. [Online; accessed 23-April-2019].

[305] Alberto Ros and Alexandra Jimborean. The entangling instruction prefetcher. *IEEE Computer Architecture Letters*, 19(2):84–87, 2020.

[306] Eric Rotenberg, Steve Bennett, and James E Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*, pages 24–34. IEEE, 1996.

[307] Shai Rubin, Rastislav Bodík, and Trishul Chilimbi. An efficient profile-analysis framework for data-layout optimizations. In *ACM SIGPLAN Notices*, volume 37, pages 140–153. ACM, 2002.

[308] J Rupley. Samsung exynos m3 processor. *IEEE Hot Chips*, 30, 2018.

[309] Ahmed Samara and James Tuck. The case for domain-specialized branch predictors for graph-processing. *IEEE Computer Architecture Letters*, 19(2):101–104, 2020.

[310] Daniel Sanchez and Christos Kozyrakis. Zsim: Fast and accurate microarchitectural simulation of thousand-core systems. In *International Symposium on Computer Architecture*, 2013.

[311] Andreas Sandberg, David Eklöv, and Erik Hagersten. Reducing cache pollution through detection and elimination of non-temporal memory accesses. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[312] Martin Schindewolf. Analysis of cache misses using simics. *Master's thesis*, 2007.

[313] J Sedlacek and H Thomas. Visualvm all-in-one java troubleshooting tool, 2018.

[314] Vivek Seshadri, Onur Mutlu, Michael A Kozuch, and Todd C Mowry. The evicted-address filter: A unified mechanism to address both cache pollution and thrashing. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 355–366. IEEE, 2012.

[315] Vivek Seshadri, Samihan Yedkar, Hongyi Xin, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, and Todd C Mowry. Mitigating prefetcher-caused pollution using informed caching policies for prefetched blocks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(4):1–22, 2015.

[316] André Seznec. Analysis of the o-geometric history length branch predictor. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 394–405. IEEE, 2005.

[317] André Seznec. Tage-sc-l branch predictors. In *JILP-Championship Branch Prediction*, 2014.

[318] André Seznec. Exploring branch predictability limits with the mtage+ sc predictor. In *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, page 4, 2016.

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

[320] André Seznec. The fnl+ mma instruction cache prefetcher. In *IPC-1-First Instruction Prefetching Championship*, 2020.

[321] André Seznec and Pierre Michaud. A case for (partially) tagged geometric history length branch prediction. *J. Instr. Level Parallelism*, 8, 2006.

[322] S Seznec. Don't use the page number, but a pointer to it. In *23rd Annual International Symposium on Computer Architecture*, pages 104–104. IEEE, 1996.

[323] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM.

[324] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. *arXiv preprint arXiv:2003.03423*, 2020.

[325] Timothy Sherwood and Brad Calder. Automated design of finite state machine predictors for customized processors. In *Proceedings 28th Annual International Symposium on Computer Architecture*, pages 86–97. IEEE, 2001.

[326] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. Applying deep learning to the cache replacement problem. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 413–425, 2019.

[327] Yannis Smaragdakis, Scott Kaplan, and Paul Wilson. Eelru: simple and effective adaptive page replacement. *ACM SIGMETRICS Performance Evaluation Review*, 27(1):122–133, 1999.

[328] Alan Jay Smith. Sequential program prefetching in memory hierarchies. *Computer*, (12):7–21, 1978.

[329] Alan Jay Smith. Cache memories. *ACM Computing Surveys (CSUR)*, 14(3):473–530, 1982.

[330] James E Smith. A study of branch prediction strategies. In *25 years of the international symposia on Computer architecture (selected papers)*, pages 202–215, 1998.

[331] Byoungro So, Mary W Hall, and Heidi E Ziegler. Custom data layout for memory parallelism. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 291–302. IEEE, 2004.

[332] Stephen Somogyi, Thomas F Wenisch, Anastasia Ailamaki, and Babak Falsafi. Spatio-temporal memory streaming. *ACM SIGARCH Computer Architecture News*, 37(3):69–80, 2009.

[333] Shixin Song, Tanvir Ahmed Khan, Sara Mahdizadeh Shahri, Akshitha Sriraman, Niranjan K Soundararajan, Sreenivas Subramoney, Daniel A Jiménez, Heiner Litz, and Baris Kasikci. Thermometer: Profile-guided btb replacement for data center applications. ISCA 2022.

[334] Shixin Song, Tanvir Ahmed Khan, Sara Mahdizadeh Shahri, Akshitha Sriraman, Niranjan K Soundararajan, Sreenivas Subramoney, Daniel A Jiménez, Heiner Litz, and Baris Kasikci. Thermometer: profile-guided btb replacement for data center applications. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 742–756, 2022.

[335] Yonghong Song and Zhiyuan Li. New tiling techniques to improve cache temporal locality. *ACM SIGPLAN Notices*, 34(5):215–228, 1999.

[336] Daniel J Sorin, Mark D Hill, and David A Wood. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 6(3):1–212, 2011.

[337] Niranjan Soundararajan, Peter Braun, Tanvir Khan, Baris Kasikci, Heiner Litz, and Sreenivas Subramoney. Pdede: Partitioned, deduplicated, delta branch target buffer. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021.

[338] Niranjan Soundararajan, Peter Braun, Tanvir Ahmed Khan, Baris Kasikci, Heiner Litz, and Sreenivas Subramoney. Pdede: Partitioned, deduplicated, delta branch target buffer. MICRO 2021.

[339] Lawrence Spracklen, Yuan Chou, and Santosh G Abraham. Effective instruction prefetching in chip multiprocessors for modern commercial applications. In *International Symposium on High-Performance Computer Architecture*, 2005.

[340] Akash Sridhar, Nursultan Kabylkas, and Jose Renau. Load driven branch predictor (LDBP). *CoRR*, abs/2009.09064, 2020.

[341] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 63–74. IEEE, 2007.

[342] Jithendra Srinivas, Wei Ding, and Mahmut Kandemir. Reactive tiling. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 91–102. IEEE, 2015.

[343] Viji Srinivasan, Edward S Davidson, Gary S Tyson, Mark J Charney, and Thomas R Puzak. Branch history guided instruction prefetching. In *International Symposium on High-Performance Computer Architecture*, 2001.

[344] Akshitha Sriraman, Abhishek Dhanotia, and Thomas F Wenisch. Softsku: Optimizing server architectures for microservice diversity@ scale. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 513–526, 2019.

[345] Akshitha Sriraman and Thomas F. Wenisch. utune: Auto-tuned threading for OLDI microservices. In *Symposium on Operating Systems Design and Implementation*, 2018.

[346] Jared Stark, Marius Evers, and Yale N Patt. Variable length path branch prediction. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 170–179, 1998.

[347] Kevin Stock, Martin Kong, Tobias Grosser, Louis-Noël Pouchet, Fabrice Rastello, J. Ramanujam, and P. Sadayappan. A framework for enhancing data reuse via associative reordering. In *Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[348] Michael Stonebraker and Lawrence A. Rowe. The design of postgres. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, SIGMOD '86, pages 340–355, New York, NY, USA, 1986. ACM.

[349] Ranjith Subramanian, Yannis Smaragdakis, and Gabriel H Loh. Adaptive caches: Effective shaping of cache behavior to workloads. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 385–396. IEEE, 2006.

[350] David Suggs, Mahesh Subramony, and Dan Bouvier. The amd "zen 2" processor. *IEEE Micro*, 40(2):45–52, 2020.

[351] Rabin A Sugumar and Santosh G Abraham. Efficient simulation of caches under optimal replacement with applications to miss characterization. In *Proceedings of the 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 24–35, 1993.

[352] Vadim Sukhomlinov and Kshitij Doshi. Selective execution of cache line flush operations, October 8 2020. US Patent App. 16/907,729.

[353] Karthik Sundaramoorthy, Zach Purser, and Eric Rotenberg. Slipstream processors: Improving both performance and fault tolerance. *ACM SIGPLAN Notices*, 2000.

[354] Herb Sutter. Eliminate false sharing. *Dr. Dobb's Journal*, (5), 2009.

[355] Masamichi Takagi and Kei Hiraki. Inter-reference gap distribution replacement: an improved replacement algorithm for set-associative caches. In *Proceedings of the 18th annual international conference on Supercomputing*, pages 20–30, 2004.

[356] M-D Tarlescu, Kevin B Theobald, and Guang R Gao. Elastic history buffer: A low-cost method to improve branch prediction accuracy. In *Proceedings International Conference on Computer Design VLSI in Computers and Processors*, pages 82–87. IEEE, 1997.

[357] Stephen J Tarsa, Chit-Kwan Lin, Gokce Keskin, Gautham Chinya, and Hong Wang. Improving branch prediction by modeling global history with convolutional neural networks. *arXiv preprint arXiv:1906.09889*, 2019.

[358] O. Temam, E. D. Granston, and W. Jalby. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Supercomputing '93:Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, pages 410–419, Nov 1993.

[359] Elvira Teran, Zhe Wang, and Daniel A Jiménez. Perceptron learning for reuse prediction. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016.

[360] Josep Torrellas, HS Lam, and John L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, 1994.

[361] Muhammed Ugur, Cheng Jiang, Alex Erf, Tanvir Ahmed Khan, and Baris Kasikci. One profile fits all: Profile-guided linux kernel optimizations for data center applications. *ACM SIGOPS Operating Systems Review 2022*.

[362] Georgios Vavouliotis, Lluc Alvarez, Boris Grot, Daniel Jiménez, and Marc Casas. Morrigan: A composite instruction tlb prefetcher. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1138–1153, 2021.

[363] Santhosh Verma, Benjamin Maderazo, and David M Koppelman. Spotlight-a low complexity highly accurate profile-based branch predictor. In *2009 IEEE 28th International Performance Computing and Communications Conference*, pages 239–247. IEEE, 2009.

[364] Vish Viswanathan. Disclosure of hardware prefetcher control on some intel processors. *Intel SW Developer Zone*, 2014.

[365] David W Wall. Predicting program behavior using real or estimated profiles. *ACM SIGPLAN Notices*, 26(6):59–70, 1991.

[366] Zhenjiang Wang, Chenggang Wu, Pen-Chung Yew, Jianjun Li, and Di Xu. On-the-fly structure splitting for heap objects. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4), 2012.

[367] Thomas F Wenisch, Michael Ferdman, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. Temporal streams in commercial server applications. In *2008 IEEE International Symposium on Workload Characterization*, pages 99–108. IEEE, 2008.

[368] Thomas F Wenisch, Michael Ferdman, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. Practical off-chip meta-data for temporal memory streaming. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, pages 79–90. IEEE, 2009.

[369] Thomas F Wenisch, Stephen Somogyi, Nikolaos Hardavellas, Jangwoo Kim, Anastassia Ailamaki, and Babak Falsafi. Temporal streaming of shared memory. In *32nd International Symposium on Computer Architecture*, pages 222–233. IEEE, 2005.

[370] Wikipedia contributors. Perf (linux) — Wikipedia, the free encyclopedia, 2018. [Online; accessed 24-April-2019].

[371] Wikipedia contributors. Vtune — Wikipedia, the free encyclopedia, 2018. [Online; accessed 23-April-2019].

[372] Wikipedia contributors. Clang — Wikipedia, the free encyclopedia, 2019. [Online; accessed 24-April-2019].

[373] Wikipedia contributors. Fowler–noll–vo hash function — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Fowler%E2%80%93Noll%E2%80%93Vo_hash_function&oldid=931348563, 2019. [Online; accessed 17-April-2020].

[374] Wikipedia contributors. Openjdk — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=OpenJDK&oldid=927329117, 2019. [Online; accessed 23-November-2019].

[375] Wikipedia contributors. Strip (unix) — Wikipedia, the free encyclopedia, 2019. [Online; accessed 24-April-2019].

[376] Wikipedia contributors. Alder lake (microprocessor) — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Alder_Lake_(microprocessor)&oldid=990207738, 2020. [Online; accessed 25-November-2020].

[377] Wikipedia contributors. Apache kafka — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Apache_Kafka&oldid=988898935, 2020. [Online; accessed 23-November-2020].

[378] Wikipedia contributors. Drupal — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Drupal&oldid=989582664, 2020. [Online; accessed 23-November-2020].

[379] Wikipedia contributors. Dtrace — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=DTrace&oldid=950798652, 2020. [Online; accessed 25-April-2020].

[380] Wikipedia contributors. Mediawiki — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=MediaWiki&oldid=989993176, 2020. [Online; accessed 23-November-2020].

[381] Wikipedia contributors. Murmurhash — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=MurmurHash&oldid=950347972, 2020. [Online; accessed 17-April-2020].

[382] Wikipedia contributors. Verilator — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Verilator&oldid=989046249, 2020. [Online; accessed 8-April-2021].

[383] Wikipedia contributors. Wordpress — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=WordPress&oldid=977243718, 2020. [Online; accessed 23-November-2020].

[384] Wikipedia contributors. Apache cassandra — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Apache_Cassandra&oldid=1010524207, 2021. [Online; accessed 7-April-2021].

[385] Wikipedia contributors. X86-64 — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=X86-64&oldid=1016690406, 2021. [Online; accessed 10-April-2021].

[386] WikiSysop. Atomic operations library. https://en.cppreference.com/w/cpp/atomic. [Online; last accessed 07-August-2018].

[387] Wayne A Wong and J-L Baer. Modified lru policies for improving second-level cache behavior. In *Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6 (Cat. No. PR00550)*, pages 49–60. IEEE, 2000.

[388] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. *ACM SIGARCH computer architecture news*, 23(2):24–36, 1995.

[389] Steven Cameron Woo, Jaswinder Pal Singh, and John L. Hennessy. The performance advantages of integrating block data transfer in cache-coherent multiprocessors. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, pages 219–229, New York, NY, USA, 1994. ACM.

[390] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C Steely Jr, and Joel Emer. Ship: Signature-based hit predictor for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 430–441, 2011.

[391] Carole-Jean Wu, Aamer Jaleel, Margaret Martonosi, Simon C Steely Jr, and Joel Emer. Pacman: prefetch-aware cache management for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 442–453, 2011.

[392] Youfeng Wu and James R Larus. Static branch frequency and program profile analysis. In *Proceedings of the 27th annual international symposium on Microarchitecture*, pages 1–11, 1994.

[393] Xiaoya Xiang, Chen Ding, Hao Luo, and Bin Bao. HOTL: a higher order theory of locality. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, pages 343–356, 2013.

[394] Zhichen Xu, James R Larus, and Barton P Miller. Shared-memory performance profiling. In *ACM SIGPLAN Notices*, volume 32, pages 240–251. ACM, 1997.

[395] Ahmad Yasin. A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 35–44. IEEE, 2014.

[396] Ahmad Yasin, Lihu Rappoport, Jared W Stark, Jeffrey Baxter, Israel Diamand, Pavel Fridman, Ibrahim Hur, and Nir Tell. Code prefetch instruction, 2021. US Patent App. 17/033,751.

[397] Tse-Yu Yeh and Yale N Patt. Two-level adaptive training branch prediction. In *Proceedings of the 24th annual international symposium on Microarchitecture*, pages 51–61, 1991.

[398] Tse-Yu Yeh and Yale N Patt. A comprehensive instruction fetch mechanism for a processor supporting speculative execution. *ACM SIGMICRO Newsletter*, 23(1-2):129–139, 1992.

[399] Adarsh Yoga and Santosh Nagarakatte. Parallelism-centric what-if and differential analyses. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 485–501. ACM, 2019.

[400] Cliff Young, Nicolas Gloy, and Michael D Smith. A comparative analysis of schemes for correlated branch prediction. *ACM SIGARCH Computer Architecture News*, 23(2):276–286, 1995.

[401] Cliff Young and Michael D Smith. Improving the accuracy of static branch prediction using branch correlation. *ACM SIGOPS Operating Systems Review*, 28(5):232–241, 1994.

[402] Tingting Yu and Michael Pradel. Syncprof: Detecting, localizing, and optimizing synchronization bottlenecks. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 389–400, 2016.

[403] Tingting Yu and Michael Pradel. Pinpointing and repairing performance bottlenecks in concurrent programs. *Empirical Software Engineering*, 23(5):3034–3071, 2018.

[404] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. Spark: Cluster computing with working sets. *HotCloud*, 2010.

[405] Siavash Zangeneh, Stephen Pruett, Sangkug Lym, and Yale N Patt. Branchnet: A convolutional neural network to predict hard-to-predict branches. In *MICRO-53*. IEEE, 2020.

[406] Yuanrui Zhang, Wei Ding, Jun Liu, and Mahmut Kandemir. Optimizing data layouts for parallel computation on multicores. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 143–154. IEEE, 2011.

[407] Yuxuan Zhang, Tanvir Ahmed Khan, Gilles Pokam, Baris Kasikci, Heiner Litz, and Joseph Devietti. Ocolos: Online code layout optimizations. MICRO 2022.

[408] Qin Zhao, David Koh, Syed Raza, Derek Bruening, Weng-Fai Wong, and Saman Amarasinghe. Dynamic cache contention detection in multi-threaded applications. In *ACM SIGPLAN Notices*, volume 46, pages 27–38. ACM, 2011.

[409] Fang Zhou, Yifan Gan, Sixiang Ma, and Yang Wang. wperf: generic off-cpu analysis to identify bottleneck waiting events. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 527–543, 2018.

[410] Jingren Zhou and Kenneth A Ross. Buffering databse operations for enhanced instruction cache performance. In *International conference on Management of data*, 2004.

[411] Yufeng Zhou, Xiaowan Dong, Alan L Cox, and Sandhya Dwarkadas. On the impact of instruction address translation overhead. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 106–116. IEEE, 2019.

[412] Craig Zilles and Gurindar Sohi. Execution-based prediction using speculative slices. In *International Symposium on Computer Architecture*, 2001.

[413] Gefei Zuo, Jiacheng Ma, Andrew Quinn, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. Execution reconstruction: Harnessing failure reoccurrences for failure reproduction. In *ACM SIGPLAN conference on Programming language design and implementation*, 2021.