SOFTWARE-ORIENTED HARDWARE PREFETCHING AND VECTOR EXECUTION

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Neil Adit

December 2024

© 2024 Neil Adit

ALL RIGHTS RESERVED

SOFTWARE-ORIENTED HARDWARE PREFETCHING AND VECTOR EXECUTION

Neil Adit, Ph.D.

Cornell University 2024

The hardware-software abstraction enables programmers to write high-level algorithms without delving into low-level microarchitectural details. Compilers, positioned at the interface of hardware and software, perform numerous optimizations to enhance performance. Nonetheless, their functionality is limited by the ISA contract designed by hardware developers. Rethinking this abstraction can unlock powerful optimizations at the compiler stage. For instance, emerging scalable vector ISAs expose hardware vector length as a programmable constant to the software, which, with compiler support, can improve vectorization opportunities in addition to code portability. Additionally, hardware prefetchers come with software prefetching knobs to leverage programmer knowledge for performance gains. However, this control is limited, unable to influence dynamic prefetching decisions made by the hardware, which has been shown to cause performance regression in datacenter settings.

This thesis aims to enhance compiler-guided optimizations for autovectorization and hardware prefetching. The auto-vectorization evaluation identifies compiler shortcomings with scalable vector ISAs, and proposes ScaleIR as a prototype, to improve mask representations in the LLVM IR. ProP uses profile-guided hints to better guide hardware prefetching decisions. Together, these projects enable compilers to effectively leverage and redefine the software-hardware abstraction, boosting performance and efficiency.

BIOGRAPHICAL SKETCH

Neil Adit was born to Manju and Sanjay Jhunjhunwala in Patna, Bihar, India (1995). He grew up in an engineering background—his father, a computer science engineer, and mother, a computer science lecturer in Patna University. Neil enjoyed access to a desktop computer at home since his childhood, which his father got quite early in 1994, as part of starting a school computer education startup. Neil was fortunate to be taught by Aftab Ahmad, a math teacher outside school from 8th to 10th grade, who introduced him to the enthralling world of geometry and number theory in a way like no one had before. This newly found interest in math propelled him towards pursuing a career in science and technology.

Neil went to the Indian Institute of Technology, Bombay for his combined bachelor's and master's degree (B.Tech + M.Tech) in Electrical Engineering. He worked with Professor Sachin Patkar for his master's thesis, on accelerating sparse matrix solvers on FPGAs using high level synthesis (HLS) tools.

Neil joined the PhD program in ECE at Cornell University in August, 2018. He was fortunate to be advised by Professor Adrian Sampson. Initially, he worked with Mark Buckler and Philip Bedoukian on different projects in the field of efficient computer vision and software-defined manycore systems. His experience from programming vector-manycore architectures, motivated him to study auto-vectorizating compilers for next-generation vector ISAs. In addition, during his graduate studies, he also got the opportunity to do summer internships at Intel Labs, Microsoft and Google. The promising results from the Google internship on hardware prefetchers, propelled him to continue collaborating and develop the prefetching work discussed in this thesis.

This document is dedicated to my family and friends.

ACKNOWLEDGEMENTS

I am grateful to all my family, friends and collaborators for supporting me throughout my graduate career. This has been a long, hard and incredibly satisfying journey, and I'm thankful to everyone I met along the way.

This thesis would not have been possible without the guidance and support of my advisor, Professor Adrian Sampson. Adrian encouraged me to approach problems with independent thought and provided invaluable, constructive feedback. After each of our meetings, I left with renewed optimism for the project—a mindset I hope to carry into my future endeavors. I am also deeply grateful to the rest of my committee—Professor Zhiru Zhang and Professor Chris De Sa—for their timely feedback and guidance throughout my graduate studies.

Before starting graduate school at Cornell, I was fortunate to be advised by Professor Sachin Patkar during my BTech & Mtech program at IIT Bombay. It was during this time that I delved deeper into hardware acceleration of sparse kernels, which ultimately inspired me to pursue PhD programs in systems and architecture.

I'd like to acknowledge everyone in our research group, CAPRA. I have always enjoyed our lunch meetings and benefited greatly from the invaluable feedback on my talks. Phil, thank you for all the jokes and guidance. I learned so much while working on projects with you, and I sincerely appreciate the time and effort you invested in mentoring me and providing feedback. Your support made a significant impact on my dissertation. Mark, you were a great mentor, collaborator and friend. I look forward to attending more of your stand-up shows and seeing the other skills you've picked up along the way.

My internship project at Google was a pivotal part of my thesis, and I was

fortunate to have two exceptional mentors—Akanksha Jain and Snehasish Kumar. Akanksha, thank you for helping me transition into a new field with ease and for providing actionable ideas that drove the project forward. You taught me to be optimistic and our one-on-one meetings were super helpful during the project. Snehasish, your expertise in compilers and profiling was crucial in navigating the project's complexities within Google. I am deeply grateful for your guidance.

I'd like to thank everyone at CSL for keeping the lab space lively and fostering a sense of camaraderie. I am grateful to our 471D group—Zhijing, Yichi, Chenhui, Peitian, Kailin, Sungbo, and Trishita—for all the fun moments and insightful research discussions. Khalid and Tuan, thank you for the gem5 deep dives. Nitish, your B-exam was incredibly motivating. Neeraj, thank you for listening to my rants and giving invaluable suggestions. Jordan, Jimmy, and Nikita, our cross-country drive needs to become a recurring tradition. Nikita, you were fantastic at spotting me and hyping me up at the gym. Varun, we briefly met in CSL/Google but it was fun hiking and dining together.

I was fortunate to have an amazing group of roommates throughout grad school. Swatah, you were a great cook and your social networking skill introduced me to so many incredible people. Apoorva, you were pretty chill and made my Seattle transition seamless. Sai, thank you for teaching me how to play Catan and booking that wonderful campsite at Vermont—we should definitely go again together! Yuktha and Shashwat, it was so much fun playing timer-driven board games, eating banana breads every Sunday and keeping the sink clean. Yousuf, please keep inviting me over for dinners.

I'd also like to acknowledge a few close friends. Jashan, thank you keeping me sane throughout grad school. I will miss our lunches at Koko, workouts at

Teagle and piano sessions at James street, but I'm sure we'll find parallels on the west coast. Pragya, I'm grateful you joined grad school—it's been wonderful having someone so relatable by my side. Reshabh, you are incredibly friendly, and I'm so glad I met you in Seattle. Shlok, our analytical discussions are a nice escape from work, and I look forward to continuing them in San Francisco. Sohan, I'm fortunate to always have you by my side, especially when I need it the most.

Finally, I would like to acknowledge my parents, whose countless sacrifices have allowed me to pursue my dreams. I am incredibly fortunate to have both of you by my side, offering unwavering support throughout my graduate studies. Mom, you have been my pillar of strength, always optimistic and holding me accountable. Your encouragement has been invaluable. Dad, your courage in facing challenges and love for science & engineering, has been a constant source of inspiration.

TABLE OF CONTENTS

Biographical Sketch . . . iii
Dedication . . . iv
Acknowledgements . . . v
Table of Contents . . . viii
List of Tables . . . x
List of Figures . . . xi

1 Introduction **1**
1.1 Vector computation . . . 3
1.1.1 Vector architectures . . . 3
1.1.2 Vector ISAs . . . 5
1.1.3 Auto-vectorizing compilers . . . 6
1.2 Hardware Prefetchers . . . 6
1.2.1 Prefetcher design principles . . . 7
1.2.2 Types of Prefetchers . . . 10
1.3 Thesis Overview . . . 12
1.4 Collaboration, Other Work, and Funding . . . 13

2 Evaluating Compiler Auto-Vectorization for RISC-V Vector **17**
2.1 Introduction . . . 17
2.2 Related Work . . . 19
2.3 Experimental Setup . . . 21
2.4 Synthetic Loop Study . . . 21
2.5 Application Benchmark Study . . . 24
2.5.1 Unmodified code . . . 26
2.5.2 Vector math libraries . . . 27
2.5.3 Vector-scalar width mismatch . . . 29
2.5.4 Dynamic vector length scalability . . . 29
2.5.5 Shuffle pattern detection . . . 31
2.5.6 Algorithm driven Loop Fusion . . . 34
2.5.7 Vectorizing specific loops . . . 35
2.5.8 Adapt algorithms to the microarchitecture . . . 36
2.6 Solution proposal: Scalable compiler IR . . . 37
2.7 Conclusion . . . 41

3 Software-Controlled Hardware Prefetching **42**
3.1 Introduction . . . 42
3.2 Motivation . . . 45
3.2.1 Reactive Throttling . . . 46
3.2.2 Predictive Throttling . . . 47
3.3 Profiling Insights . . . 49
3.3.1 Program counter maps to distinct prefetching behavior . . 50

3.3.2 Code context and data features can enhance prefetching understanding . . . 54
3.4 Programmable Prefetching . . . 56
3.4.1 Profiling Analysis . . . 56
3.4.2 Communicating Prefetch Hints to Hardware . . . 64
3.4.3 Prefetch Filtering in Hardware . . . 67
3.5 Methodology . . . 67
3.5.1 Performance Model . . . 67
3.5.2 Workloads . . . 68
3.5.3 Profiling methodology . . . 70
3.5.4 Baselines . . . 71
3.6 Evaluation . . . 73
3.6.1 Single core results . . . 73
3.6.2 Multi-core results . . . 76
3.6.3 Context-switch sensitivity . . . 77
3.6.4 Bandwidth sensitivity . . . 81
3.6.5 Workload-specific profile-model tuning . . . 82
3.6.6 Different LLC sizes . . . 83
3.7 Related Work . . . 84
3.8 Conclusion . . . 86

4 Conclusion and Future Directions **87**
4.1 Length-agnostic speculative vectorization . . . 87
4.2 Programming model for scalable vectors . . . 90
4.3 Multiple hardware prefetchers . . . 91

Bibliography **93**

LIST OF TABLES

2.1 We propose solutions for compiler auto-vectorization issues and rate the difficulty from a compiler's standpoint, ranging from well-defined engineering fixes (E) to compiler (C) and programming model (P) research problems. The proposals are grouped (A,B) based on the two evaluation benchmarks. 19
2.2 RiVEC benchmark transformations to aid compiler auto-vectorization for an objective performance measurement 24

3.1 Simulation configuration 67
3.2 Workload simulation details 68

LIST OF FIGURES

1.1 Hardware prefetchers (shown in boxes) trade off higher cache hit rates (higher coverage) for higher memory bandwidth overhead (lower accuracy). The points roughly represent the default parameter configurations of prefetchers shown in Fig 3.2. 8

2.1 Auto-vectorized TSVC loops for RVV-VLA and RVV-VLS configurations 22
2.2 Dynamic instruction count speedup over the scalar version using a vector length of 8. Transformations for the auto-vector configurations are based on Table 2.2, whereas the serial and RiVec versions are transformed to skip math functions only. 25
2.3 Dynamic instruction scaling across different hardware vector length in Jacobi-2d plotted on logscale. The overhead of running scalar instructions (due to non-scalability) increases at higher vector length for compiler generated code. 31
2.4 Vector memory requests relative to hand-vectorized baseline. The transformation allows compiler to reduce redundant loads in both the auto-vectorized configurations. 33
2.5 Instruction selection procedure in LLVM vectorization for fixed vector-length configuration of RVV. The LLVM IR represents the data movement across vector registers using the `shufflevector` intrinsic using a fixed vector-length mask array. This representation fails for length-agnostic designs. 37
2.6 Represent shuffle masks using a function of vector-id and scalable vector length in the IR. 39

3.1 The benefit of hardware prefetchers shrink with limited bandwidth on both Merced (a large datacenter workload) and mcf (SPEC workload). 46
3.2 Accuracy vs coverage tradeoff for different configurations of IP-stride, Best Offset (BOP) and Signature Path (SPP) prefetchers, on `Merced` and `Bravo` datacenter workloads. 47
3.3 Context switches are frequent in datacenter workloads, making it difficult for online throttling schemes to warm up adequately. . 48
3.4 Prefetching accuracies with BOP for each PC in a code region. . . 49
3.5 PC-based SPP prefetcher accuracy histogram, for different accuracy bin sizes. A significant portion of prefetching is done for PCs that are < 30% accurate. PCs themselves, can be a good indicator for prefetching behavior since frequencies of < 10% and > 90% bins are highest. 51
3.6 PC-based BOP prefetcher accuracy histogram, for different accuracy bin sizes. 51

3.7 Aggregated SPP prefetcher accuracies for various libraries called during the `Merced` trace execution. `foo` is an anonymized library name. 52
3.8 Function-level breakdown of prefetching accuracies 52
3.9 Expected prefetch accuracy if low accuracy (< 30%) features are filtered out. "No filter" is the baseline prefetcher accuracy. PC, Page and Call stacks (CS) are features that can be used independently or hierarchically to improve overall prefetch accuracy. . . 55
3.10 An overview of the Programmable Prefetching (ProP) system. . . 57
3.11 Fraction of miscategorized PC (%) based on total prefetches issued, decrease with additional level of microarchitectural modeling. The ground truth prefetch accuracy data for PCs is from gem5 simulation of `Merced`. 60
3.12 Fraction of miscategorized PC (%) by the profiling model for all datacenter workloads, for different underlying hardware prefetchers. The miscategorization error is small in all workloads, and generalizes well to both hardware prefetchers 62
3.13 Cumulative density function (CDF) of number of unique PCs (in log scale) needed to represent prefetch behavior of datacenter workloads. 65
3.14 Single core IPC improvement over no prefetching for all datacenter workloads at 4GB/s bandwidth 72
3.15 Accuracy-Coverage tradeoff of various prefetcher configurations, averaged across all datacenter workloads. ProP-BOP and ProP-SPP allows better coverage and accuracy tradeoff, and low traffic overhead. 74
3.16 Library-level prefetch accuracies with baseline BOP and ProP-BOP. 75
3.17 Single core IPC improvement over no prefetching for SPEC workloads at 4GB/s bandwidth 76
3.18 IPC speedup for 100 datacenter workload mixes in a 4-core setting at 16GB/s system bandwidth. 77
3.19 IPC speedup for 100 datacenter workload mixes in a 8-core setting at 32GB/s system bandwidth. 78
3.20 IPC speedup averaged for all datacenter workloads, simulated at different instruction window lengths. 79
3.21 Prefetch traffic and prefetch filtering efectiveness of different prefetchers at different instruction window sizes. 79
3.22 IPC improvement (%) of ProP-SPP over PPF-SPP across different instruction windows. 80
3.23 IPC improvement over no prefetching on all datacenter workloads, for SPP, PPF and ProP at different system bandwidths. . . 81
3.24 Workload-specific speedup for different prfetchers and profiling thresholds of ProP (default, aggressive, conservative), across bandwidth configurations. 82

3.25 IPC improvement over no prefetching on all datacenter workloads, for SPP, PPF and ProP at different SLC sizes/core. 83

CHAPTER 1

INTRODUCTION

Hardware and software systems have evolved in complexity over the last century. Emerging applications demand high computation density, which has been fueled by Moore's law (transistor doubling every two years) and Dennard scaling (increasing clock frequency without increasing power consumption) over the years. However, as these scaling laws have either stopped or slowed down considerably, there has been interest in rethinking the hardware-software abstractions to improve performance and energy efficiency.

The role of compilers, for instance, has shifted from the early years of computing when they were primarily tools for translating high-level code into machine code, with the primary goal of ensuring correctness and basic optimizations. Over the years, as the gains from Moore's Law began to diminish, compilers became critical for squeezing out performance improvements through more aggressive optimizations, such as loop unrolling [116], vectorization [71, 51], and profile-guided optimizations [80, 32]. In addition, with slowdown of Dennard scaling and power efficiency becoming critical, specialized architectures such as GPUs, FPGAs, TPUs and other domain-specific accelerators have been widely adopted. This has led to an explosion in compiler toolchains [73, 119, 70] targeting specialized architectures and domains, while exposing hardware optimizations at the software level.

There are several computing domains where hardware-software co-design techniques have been incorporated to improve performance and energy efficiency such as custom accelerators for machine learning [20, 45], network processing [53] etc. In addition, general purpose micro-architectural components

such as branch prediction and cache management have also seem improvements with software-guided approach. Profile-driven compiler hints about branch outcomes have been shown to improve hardware branch prediction accuracy [9, 59, 47]. Similarly, compilers can provide hints to cache controllers about evictions of cache lines ahead of time [114, 99]. In this thesis, we'll focus on two such domains that can benefit from software-driven co-design optimizations.

There has been a resurgence in interests for vector architectures, led by length-agnostic ISAs such as ARM's Scalable Vector Extension (SVE) [109] and RISC-V Vector Extension (RVV) [94]. These scalable vector ISAs aim to rethink the existing contract between hardware and software, to improve programmability and performance of vector architectures. Vector support in hardware allows programmer to exploit data-level parallelism (DLP) in the program, in addition to hardware support for instruction-level parallelism (ILP) via out-of-order cores. This software-hardware interaction offloads the overhead of finding parallelism and coalescing memory accesses of embarrassingly-parallel workloads with DLP to the compiler, while allocating more compute on-chip. This leads to an overall improvement in performance and energy efficiency, over a design with minimal communication between these abstraction layers.

Similarly, hardware prefetchers have a long line of work [103, 44, 7, 75, 19, 31, 50, 98, 69, 106, 21, 117, 105, 118, 22, 40, 41, 88, 100, 62, 48], where sophisticated prefetcher designs have worked well in reducing memory cache misses and thereby improving performance on single-core machines. In addition, there has been a surge in interest towards improving accuracy of hardware prefetchers to perform well in multi-core settings with limited memory bandwidth [113, 107,

49, 11, 74, 76]. However, pure hardware prefetching designs may be reaching a limit since they focus on what addresses to prefetch (performance-centric) *and* which accesses to prefetch on (accuracy-centric), which can require high on-chip memory and longer warm-up times. We propose that a SW/HW co-design of prefetchers, which divides responsibilities across the stack, can enable them to have *both*—high performance and memory bandwidth efficiency.

1.1 Vector computation

Vector abstractions enable aggregating the same function over multiple data elements and if efficiently exploited, leads to significant speedup over scalar computation, for DLP-rich applications. In this section, we discuss three layers of the software/hardware stack that affects the overall efficiency of vector processing, and motivate the need for improved compiler techniques to leverage emerging vector architectures and ISAs.

1.1.1 Vector architectures

Vector architectures are geared towards exploiting data parallelism in workloads by loading chunks of data from memory and performing the same function over each of them. To this end, a vector architecture typically has three major components, in addition to a scalar processor:

- Vector memory unit: Performs grouped load/store accesses for different memory patterns such as contiguous or strided accesses.
- Vector registers: Exploit temporal reuse of data similar to scalar registers.

- Vector functional units (VFUs): Supports processing multiple elements in parallel lanes.

Traditionally, there are two categories of vector architecture designs: long vector-length decoupled architectures and shorter vector-length packed-SIMD architectures. The former optimizes high vector performance at the cost of larger on-chip area, whereas the latter is tradeoff for more efficient scalar compute with modest vector performance, achievable with a scalar-like code.

Long vector architectures: The earliest vector architectures [115, 36] were memory-to-memory vector machines (no vector registers), which could support an arbitrary vector length. However, they had poor scalar performance and vector computations needed long startup times. Cray-1 [96] integrated vector registers and VFUs with scalar computation, while retaining long vector length support using chiming. Since then, long vector architecture designs [27, 2] have added support for complex memory accesses, and work well with scalar out-of-order cores and multi-core processors.

Packed-SIMD: Packed-SIMD [77, 25, 38, 5] architectures are typically smaller vector units that have a 1:1 mapping of vector register elements and functional unit lanes. These architectures tradeoff some vector performance for lower area overhead. Packed-SIMD architectures support cross element functions such as vector shuffles and reductions, due to small number of lanes. However, they typically avoid supporting additional control overhead for masking lanes which requires scalar processing of loop tails.

Next-generation vector architectures lie on this tradeoff space of long-vector

performance and packed-SIMD area overhead. For instance, reconfigurable designs [10, 111, 120] can functionally mimic large vector processors while maintaining minimal area overhead like packed-SIMD architectures.

1.1.2 Vector ISAs

There are two classes of vector ISAs: (1) *fixed vector length ISAs*, designed for packed-SIMD architectures that have a fixed vector length; and (2) *scalable vector ISAs*, designed for a more expansive set of vector architectures, and inspired by traditional architectures such as Cray-1 [96], which supports variable vector lengths via chiming.

Fixed vector length ISAs: Fixed vector length ISAs [77, 38, 5], or SIMD ISAs, bake the hardware vector length in the instructions and often do not provide predicate-masking support to use a "smaller" vector length. SIMD instructions often require extensive loop unrolling to keep hardware pipelines busy but puts additional pressure on instruction caches. No predication also incurs an additional overhead of handling scalar loop tails.

Scalable vector ISAs: Emerging scalable vector ISAs [109, 94] abstract the vector length out of the instruction and let the programmer tailor the effective vector length. For instance, RISC-V Vector extension (RVV) [94] provides an interface to configure the hardware vector length using `vsetvl` instruction. Arm Scalable Vector Extension (SVE) [109] uses predicate-based loop directives such as `whilelt` to dynamically configure vector lanes and enable loop tail vectorization. These "scalability" features have been shown to improve vector utilization

by providing finer-grained control of vector length and supporting predicate-based vectorization of complex control-flow, in addition to better portability across machines with different vector lengths [109].

1.1.3 Auto-vectorizing compilers

Compiler toolchains such as GCC [34] offer support for automatic vectorization [33] to exploit DLP in programs, as an alternative to manual intrinsic-based programming. Loop vectorization and SLP vectorization [51, 86, 87, 61] are the two main compiler vectorization techniques. Since loops express parallelism over variable loop bounds, they provide better vectorization opportunities for scalable vector lengths. In the past, various improvements to loop vectorization have been proposed to identify interleaving memory patterns [71], vectorize outer loops [72], and loops with control flow [110, 8].

These vectorization techniques were developed for fixed vector-length ISAs, where vector length is known at compile time. Next-generation vector architectures and scalable ISAs pose a challenge for compiler techniques to improve code-generation for dynamic vector length and support complex memory gather/scatter and predicate instructions that were restricted in early packed-SIMD designs.

1.2 Hardware Prefetchers

Hardware prefetchers hide memory latency by fetching a memory block in anticipation of the program accessing it in the near future. Since memory latencies

are much higher compared to processor speeds, accurately fetching blocks beforehand can reduce backend memory-bound stall cycles. However, prefetching a future block consumes memory bandwidth that also needs to service the current miss. Fundamentally, this creates a tradeoff between prefetcher coverage and traffic overhead. Hence, an aggressive prefetcher with complex strategies could end up increasing memory bandwidth contention and overall memory latency in a bandwidth-constrained environment, but work well if lots of spare bandwidth is available. We discuss design principles and existing types of prefetchers, to understand tradeoffs in hardware prefetchers.

1.2.1 Prefetcher design principles

Prefetcher designs have revolved around three fundamental questions: 1. *what* addresses to prefetch, 2. *when* to initiate prefetch requests and 3. *where* to place the prefetch data.

***What* addresses to prefetch.** This decision can influence metrics such as cache miss rate, bandwidth usage and cache pollution. Cache miss rate reduction is defined by *coverage*, which is the fraction of demand misses that were prefetched by the hardware prefetcher. On the other hand, bandwidth consumption depends on prefetcher *accuracy*, which is defined as fraction of used prefetches over total prefetches issued. A poor prefetcher accuracy results in an increased bandwidth consumption. Various prefetching algorithms have been proposed to determine which addresses to prefetch based on a history of demand accesses, ranging from instruction-pointer stride (IP-stride) prefetcher [7] to sophisticated irregular pattern-predicting prefetchers [48]. Ideally, hardware

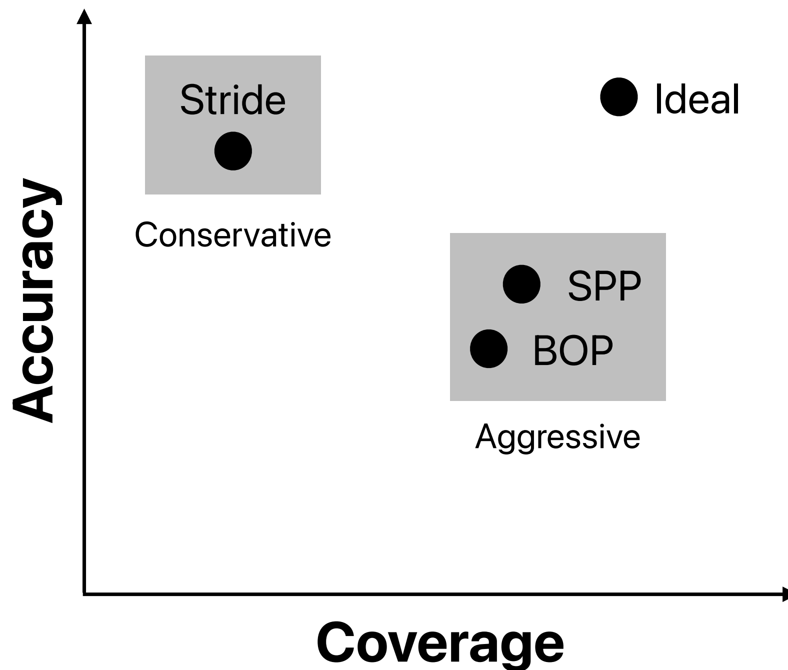


Figure 1.1: Hardware prefetchers (shown in boxes) trade off higher cache hit rates (higher coverage) for higher memory bandwidth overhead (lower accuracy). The points roughly represent the default parameter configurations of prefetchers shown in Fig 3.2.

prefetchers would have *both*: high coverage and high accuracy. However, Figure 1.1, which approximately represents data from Chapter 3, conveys that recent prefetcher designs such as Best Offset Prefetcher (BOP) [62] and Signature Path Prefetching (SPP) [48] tend to be aggressive and tradeoff prefetcher accuracy for higher prefetcher coverage.

***When* to initiate prefetch requests.** A prefetcher can choose to prefetch an address upon *all* demand accesses or just a subset of those. Prefetching on all accesses can result in excess traffic. Tagged prefetching [35] showed that prefetching on demand misses or demand hits due to previously prefetched blocks, can get similar coverage as prefetching on all accesses without the excess traffic

overhead. In addition, the prefetcher also decides how far ahead to prefetch a data address. Prefetching the "next" element in the stream might not be *timely* enough to hide the entire memory latency. Stream buffer [44] improved timeliness by prefetching multiple lines ahead. Similarly, IP-stride [7] and SPP [48], have lookahead degrees to prefetch further into the stream. BOP uses a cache-fill notification-based feedback mechanism to tune offsets to make prefetching more timely. Since prefetching ahead comes at a cost of lower prefetching confidence, it also leads to lower prefetcher accuracies.

***Where* to place the prefetch data.** Prefetchers can either prefetch directly into caches or a privately held buffer. Stream buffers [44] introduced additional buffers for different streams with lookahead data addresses to ensure timeliness whithout polluting the cache. However, to avoid the extra hardware cost and complexity in memory system, prefetchers tend to prefetch directly into cache. This can result in cache pollution if the prefetcher is not accurate. To avoid this issue, prefetchers either throttle prefetching (BOP turns off prefetching below a bad score) or prefetch at higher level caches that are larger (SPP prefetches into L2/LLC based on confidence thresholds) to avoid polluting the cache.

Improving prefetcher accuracy without decreasing coverage or increasing hardware complexity, can resolve issues across all the 3 fundamental design principles of prefetchers and move state-of-the-art prefetchers closer towards an ideal design, as shown in Figure 1.1.

1.2.2 Types of Prefetchers

Coverage-optimized hardware prefetchers: These prefetcher designs tend to optimize for coverage by improving memory access pattern detection.

Sequential-line prefetching [102], or one block lookhead (OBL) [103] are simple prefetcher designs that prefetch the next block $i + 1$ upon access of block i. Stream buffers [44] introduced separate buffers per stream to store lookahead prefetches for improved timeliness (better latency hiding) and avoiding cache pollution. Further improvements to stream buffers added support for non-unit strides [75], improved stride prediction using confidence counters [19] and Markov predictors [98], spatial bitmaps for recurring spatial locality [50, 106] and support for more complex stride patterns [40, 100, 48].

Bandwidth-efficient hardware prefetchers: As discussed in Section 1.2.1, prefetcher designs have a tradeoff between coverage and accuracy (or bandwidth overhead). This class of prefetchers primarily focus on improving accuracy of the *underlying* prefetchers. Typically there 2 sub-classes: (1) throttle prefetchers, which uses a feature such as stream length or prefetch accuracy in a region, to reactively filter prefetches; (2) composite prefetchers consists of accurate sub-prefetchers which detect and prefetch a narrow class of memory access patterns.

1. Throttle prefetchers: Adaptive stream detection (ASD) [37] measures aggregated stream length histogram (SLH) for all streams in a program region. ASD uses SLH to stop prefetching if the likelihood of the stream ending is high, to avoid prefetching a useless address. FDP [107] uses dy-

namic metrics such as prefetcher accuracy and cache pollution, to control aggressiveness of the prefetcher. For instance, if the prefetcher accuracy is below a threshold, FDP throttles the stream prefetcher by decreasing its degree and lookahead distance. Unlike the previous two uniform throttling mechanisms, Perceptron filter (PPF) [11] is a hardware-based fine-grained prefetch filtering mechanism for Signature Path Prefetcher (SPP). PPF trains a perceptron model using a set of nine features based on page addresses, program counter and other SPP metrics such as confidence, and filters SPP candidates if the perceptron weights are below a threshold.

2. Composite prefetchers: Unlike filtering techniques that aim at improving accuracy of the underlying state-of-the-art prefetcher, an alternate bandwidth-efficient design is to use a mix of simpler prefetchers for different memory access patterns. Division of Labor (DOL) [49] detects strided instructions by identfying loops, and pointer chains based on program semantics, and uses this classification to prefetch via specialized prefetchers. Instruction pointer classifier-based prefetching (IPCP) [74] uses confidence of a sub-prefetchers to classify memory accesses for each instruction pointer at L1D. Both these work, reflect the importance of identifying memory accesses that work best with the corresponding hardware prefetcher to improve accuracy and coverage.

Software-guided hardware prefetchers: Unlike previous classes of pure hardware-based prefetching, this class uses hints from the compiler to guide hardware prefetching decisions.

Guided-region prefetching (GRP) [113] uses compiler static-analysis to detect loop bounds of load instructions, and classify instructions based on memory

access patterns. The loop bounds, if statically discernible, is used to guide the underlying region prefetcher [56] to limit the size of prefetching region, thereby reducing traffic overhead. Efficient content-data prefetching (ECDP) [26] uses profile-guided hints to score all possible prefetch offsets in a physical page, for each pointer-based instruction. ECDP communicates a 16-bit one-hot encoded prefech hint with each pointer instruction, to filter prefetches by the underlying linked data structure (LDS) prefetcher [23].

1.3 Thesis Overview

Co-designing hardware-software systems together provides additional opportunities of exploiting performance and efficiency. In case of emerging vector architectures, new ISA designs provide software abstractions that compilers can exploit for new vectorization avenues. On the other hand, hardware prefetcher designs are limited by on-chip cost and limited view of the program, and can leverage higher efficiency with software hints and programmability. This thesis discusses our work on using compiler and profiling techniques to improve performance on vector architectures and hardware prefetchers, detailed in the following two chapters.

Chapter 2 explores the auto-vectorizing ability of compilers for emerging scalable vector ISAs. It identifies issues with compiler toolchains like LLVM, which have mature vectorization passes for fixed-length ISAs, but do not perform well with scalable vector ISAs. To this end, we make the following contributions: (1) we distill a set of recommendations aimed at improving compiler design, based on our evaluation across a set of synthetic loop benchmark and

real benchmark suite with hand-vectorized code for comparison; and (2) propose `ScaleIR` to improve vector representations in the compiler IR stage, which leads to better instruction selection in the backend.

Chapter 3 details the problem of hardware prefetchers consuming high bandwidth, thereby adversely affecting performance in multi-tenant datacenter setting. One solution for high accuracy is to rely on software prefetching, but its dependence on programmer or compiler guidance hurts it's universal adoption. An alternative solution is hardware-based throttling mechanisms that can identify bad regions or accesses for prefetching. However, these solutions tend to be reactive and workload-sensitive, rendering poor results at a datacenter scale. We explore a design point with a hardware-software division, where software determines which *code* to prefetch and hardware is responsible for which *data* to prefetch for it, and when. To this end, we make the following contributions: (1) we identify and evaluate issues with existing hardware throttling mechanisms, (2) evaluate efficacy of code-based features such as Program Counter (PC) for identifying prefetch usefulness for datacenter workloads, and (3) propose a Programmable Prefetching (ProP) paradigm to use profile-guided hints to improve performance of various state-of-the-art hardware prefetchers in bandwidth constraint settings.

1.4 Collaboration, Other Work, and Funding

I led the projects described in Chapter 2 and Chapter 3, while collaborating with various people.

Tuan Ta, Khalid Al-Hawaj, Professor Christopher Batten and Professor

Adrian Sampson contributed to Chapter 2. Tuan Ta and Khalid Al-Hawaj provided us with the gem5 implementation of RISC-V vector (RVV) extension. In addition, Christopher Batten, Khalid Al-Hawaj, and Tuan Ta, provided us valuable feedback and discussion. I extended the public benchmark suite for compatibility with latest compiler autovectorizer, extended gem5 to increase support for RVV instructions emitted by the compiler, modifed compiler IR and implemented backend passes for LLVM. The compiler evaluation work was published in IEEE Mirco Special Issue, 2022 [3].

Chapter 3 was performed in collaboration with Google. Akanksha Jain, Snehasish Kumar and Professor Adrian Sampson contributed to this work. Akanksha Jain provided insights into existing hardware prefetcher issues at datacenter scale, and ideas for profiling algorithms to determine prefetch hints. Snehasish Kumar gave valuable feedback on profiling datacenter workloads, and evaluating different code and data features to enhance prefetch understanding. I implemented profiling models to generate software hints, designed the architecture and extended gem5 modeling for the final evaluation.

I contributed to a few other projects in addition to the work described above.

I worked on an activation pruning technique [15], led by Dr. Mark Buckler, to retain dense computation while achieving speedup on efficient CNNs like MobileNetV2. This work proposed pruning *activation* inputs to the 1×1 convolutions using Discrete Cosine Transform (DCT). Individual channel sensitivity to frequency pruning varies, but is monotonic with respect to frequency. This property allows us to perform fine-grained dense pruning in the network and achieve speedup with minimal accuracy reduction. I implemented a training method to learn the pruning masks while refining the network accuracy, *alterna-*

tively. We demonstrate the method on CIFAR100 and ImageNet, and show improved compression with lower accuracy degradation than other related pruning works.

Dagger [52, 53], led by Nikita Lazarev, offloads the entire RPC stack on a FPGA-based NIC and leverages tight CPU-NIC coupling using NUMA interconnects. I led the effort to improve RPC throughput between the CPU and NIC interface. To this end, we enable parallel RPC requests on multiple cache lines of the shared interconnect and use a load balancer to schedule incoming RPCs. Dagger is able to achieve the lowest median round-trip latency and a high throughput of RPC requests among existing RPC acceleration fabrics.

Software-defined vectors (SDV) [10], led by Dr. Philip Bedoukian, explores dynamically reconfiguring manycore tiles into vector groups to bridge the gap between MIMD and SIMD architectures on the same fabric. We leverage a decoupled access/execute (DAE) scheme to centralize wide coalesced loads using a scalar core leading a vector group. I worked on formalizing synchronization guarantees for prefetching accesses by the decoupled run-ahead scalar core, into the scratchpad of vector cores. SDV achieved a speedup of 1.7× over optimized MIMD execution while saving 22% energy.

This thesis describes research supported in part by the Intel and NSF joint research center for Computer Assisted Programming for Heterogeneous Architectures (CAPA), in part by the NSF under Awards 1845952 and 1723715, and in part by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under Grant Agreement FA8650-18-2-7863. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and

conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

CHAPTER 2

EVALUATING COMPILER AUTO-VECTORIZATION FOR RISC-V VECTOR

2.1 Introduction

Data-parallel code is abundant in scientific computing and ML driven applications. To take advantage of regular computation, hardware units are increasingly adopting vector processing. However, there exists a gap between programming applications with minimal hardware primitives and compiling them down to efficiently utilize hardware resources.

Commercial hardware like GPUs “bridge” this gap by using an implicit vectorization-based programming model and offloading the vectorization efforts to the hardware. GPUs use a SPMD (Single Program Multiple Data) programming model, which allows the programmer to write thread-level parallel code for a *single core* and dynamically use vector instructions. However, this programmability comes at a cost of dedicated hardware to dynamically coalesce memory requests.

Traditional SIMD-style vector processors are programmed using two ways: manually using intrinsics and compiler auto-vectorization. Compiler auto-vectorization [51, 71, 72] has been well studied to alleviate the effort of manual programming of vector processors. However, programming models and compiler support for vector processors are still evolving. The need for improved auto-vectorization is accentuated due to constantly changing hardware (increasing vector length, newer vector instructions) and newer applications.

Typically, vector ISA extensions in the past have evolved in lockstep with the hardware, which requires reprogramming efforts for new vector design. A new wave of *length-agnostic* vector ISAs, led by ARM SVE [109] and RISC-V's vector extension [94] addresses this issue by decoupling the ISA from the hardware. This introduces newer challenges to the fundamental designs of the compiler toolchain built around fixed-length extensions and requires a rethinking for programming scalable vectors.

This chapter seeks to understand how auto-vectorizing compilers need to evolve to fully exploit length-agnostic ISAs. We perform two empirical evaluations to study compiler auto-vectorization in the context of the RISC-V vector (RVV) extension and the LLVM compiler infrastructure. First, we use a set of synthetic loops with broad coverage to identify auto-vectorization differences between fixed-length and length-agnostic ISAs. Next, using a set of data-parallel applications with hand-vectorized implementations, we measure the performance gap between intrinsic-based programming and compiler auto-vectorization configurations. To further understand this gap, we transform the applications' scalar code to model improvements in the compiler and programming model, and measure their impact on closing-in the gap.

We compile a list of proposals in Table 2.1, based on the issues we find from both the evaluations and estimate the difficulty of each proposal. We see these potential improvements as an outline for future work on rethinking auto-vectorization in the context of RISC-V and other scalable vector ISAs.

Table 2.1: We propose solutions for compiler auto-vectorization issues and rate the difficulty from a compiler's standpoint, ranging from well-defined engineering fixes (E) to compiler (C) and programming model (P) research problems. The proposals are grouped (A,B) based on the two evaluation benchmarks.

	Proposals	Difficulty
A	Standardize IR representation (C)	⋆⋆⋆
	Runtime vector-length based analysis (E)	⋆
	Multi-length SLP vectorization (E)	⋆⋆
	Vector reduction in dynamic loop (E)	⋆
B	Math library vectorization for RISC-V (E)	⋆
	Infer scalar width from vector code (C)	⋆⋆
	Dynamic vector length scalability (C)	⋆⋆⋆
	Shuffle pattern detection (C)	⋆⋆⋆
	Algorithmic loop fusion (P)	⋆⋆⋆⋆⋆
	Vectorizing specific loops (C,P)	⋆⋆⋆⋆
	Tune algorithm to μarch (P)	⋆⋆⋆⋆⋆

2.2 Related Work

Parallel processing. There are two opposing styles for parallel processing: MIMD-based manycores [24, 68] and SIMD-based vector processors [97, 28]. Software Defined Vectors [10] tries to bridge this gap using reconfigurable vector groups on a manycore fabric. An emerging trend of scalable vector processors and manycore architectures demands better compiler and programming support for widespread accessibility.

Vector compiler evaluation. Maleki et al. [60] evaluated compiler auto-vectorization in GCC, Intel C Compiler (ICC) and IBM's XLC compilers for 128-bit fixed-length vector ISAs and proposed an extended version to the original TSVC (Test Suite for Vectorizing Compilers) benchmark [16]. Subsequently, additional compiler evaluations [90, 63, 101] have focused on advanced fixed-vector extensions: AVX2 and AVX-512. Prior work on evaluating next-generation vector compilers [85, 83] has focused on comparing ARM SVE with

ARM Neon and Intel AVX fixed-length ISAs. The prior work's focus on comparing between compilers leads them to focus only on the loops that are feasible to vectorize with current compilers. In this chapter, we evaluate RISC-V and our goal is not only to bridge the gap between fixed- and scalable-vector designs, but also to understand the remaining gap with hand-vectorized code.

Auto-vectorization. Loop vectorization and SLP vectorization [51, 86, 87, 61] are the two main compiler vectorization techniques. Since loops express parallelism over variable loop bounds, they provide better vectorization opportunities for scalable vector lengths. In the past, various improvements to loop vectorization have been proposed to identify interleaving memory patterns [71], vectorize outer loops [72], and loops with control flow [110, 8]. These techniques have revolved around traditional fixed vector-length ISAs. We explore loop optimization opportunities in context of vector length-agnostic designs.

Vector programming models. Higher level algorithmic optimizations are out of scope for compiler vectorization and provide an opportunity for vector programming models in higher-level languages [93, 95, 29, 30, 81, 55] and DSLs that compile to vector ISAs [89]. Pohl et al. [84] evaluated existing vector programming models in C++ and showed a wide gap between them and manual-intrinsic programming. In addition, programming models tend to bake the vector length requirements in the library making them incompatible for length-agnostic ISAs. An improved model in context of length-agnostic design, along with compiler modifications can enhance the way scalable vectors are programmed.

2.3 Experimental Setup

Our experiments use a recent source version of LLVM Clang 15.0.0 [57]. We evaluate auto-vectorization for RVV and AVX-512 extensions. All configurations - scalar, hand-vector and auto-vector are compiled with `-Ofast` flag, which enables math library approximations in addition to `-O3` optimizations. The scalar and hand-vector configurations are compiled with `-fno-vectorize, -fno-slp-vectorize` to disable any compiler auto-vectorization. The auto-vector configuration for AVX-512 is also compiled with `-fveclib=libmvec` which allows LLVM to vectorize math lib calls using GLIBC vector math library.

Auto-vectorized versions can have three configurations: RVV Vector Length Specific (RVV-VLS), RVV Vector Length Agnostic (RVV-VLA) and Intel AVX-512. The LLVM compilation flag for VLA is `-scalable-vectorization=on`, VLS is `-riscv-v-vector-bits-min=N`, where *N* determines the fixed vector width and AVX-512 is `-mavx512f -mavx512cd` which enables fixed vector length of 512 bits.

We extend the gem5 simulator [13] to support RISC-V vector instructions to evaluate performance. We use the Atomic CPU model to measure dynamic instruction based statistics. We profile dynamic instruction count for AVX-512 natively on Intel Xeon Gold 6230 using `perf` and compare the auto-vectorzied instruction speedup to RISC-V counterparts.

2.4 Synthetic Loop Study

We first study the breadth of LLVM's support for auto-vectorization for RISC-V using TSVC benchmark. We compile all 151 loops from TSVC benchmark and

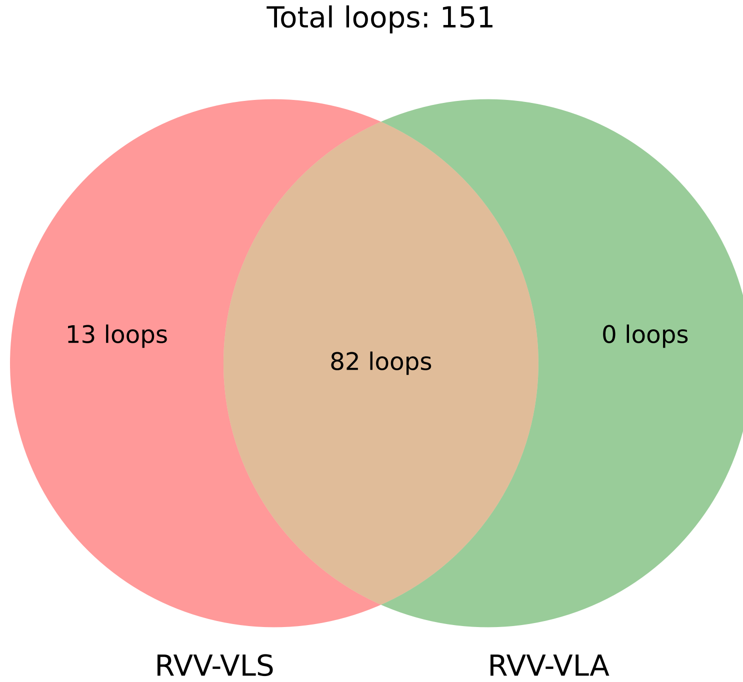


Figure 2.1: Auto-vectorized TSVC loops for RVV-VLA and RVV-VLS configurations

measure auto-vectorization differences between RVV VLS and VLA configurations. We use the instruction count speedup as a metric for compiler vectorization performance and define it for a configuration, c as:

$$\text{speedup}_c = \frac{\text{Dynamic instruction count of scalar config}}{\text{Dynamic instruction count of config, } c} \tag{2.1}$$

Figure 2.1 shows that RVV-VLS auto-vectorizes 13 loops in addition to 82 loops vectorized in both configurations. Among the 82 commonly vectorized loops, RVV-VLS and RVV-VLA have a geometric average of 7× and 6.3× instruction count speedup respectively over the scalar version for a vector length of 8, but have a few loops with differences in instruction selection. In addition, 13 loops are only auto-vectorized in RVV-VLS configuration because they

need compile-time fixed vector length for vectorization passes. We discuss these cases below and propose relevant solutions, also summarized in Table 2.1(A):

- **Instruction selection differences**: VLS configuration can select strided loads (`vlse`) whereas VLA relies on the more general indexed loads (`vluxei`) for memory access pattern like below:

  ```
  for(int i = 0; i < N; i+=2){
    a[i] = a[i - 1] + b[i];
  }
  ```

  This is due to an underlying compiler representation issue for length agnostic ISAs. In general, the offsets of a gather instruction and shuffle masks cannot be represented as a numerical array since vector length is unknown for VLA, which hampers the backend instruction selection procedure.

  Standardize IR representation and backend passes for gather offsets and shuffle masks to be length agnostic.

- **Loop carried dependence analysis**: To vectorize a loop, dependence width should be greater than vector length. However, vector length is unknown for VLA at compile-time but could be speculated [110].

  Dynamically check hardware supported vector length to conditionally execute vector code.

- **SLP vectorization**: Merging fixed number of instructions based on vector length.

  Emit SLP vectorized code for cost-effective vector widths and dynamically execute one of them based on hardware vector length.

Table 2.2: RiVEC benchmark transformations to aid compiler auto-vectorization for an objective performance measurement

Name	Suite	Transformations
Blackscholes	PARSEC	Skip Math function
Canneal	PARSEC	Loop fusion
Jacobi-2D	PolyBench	Restrict to non-aliasing memory; Simplify 2D access
Pathfinder	Rodinia	Restrict to non-aliasing memory; Simplify memory access pattern
Particle Filter	Rodinia	—
Streamcluster	PARSEC	—
Swaptions	PARSEC	Skip Math function; Inline function calls; Loop interchanging

- **Product reductions**: Final reduction across vector register needs to be unrolled by the factor of vector length.

  Perform vector register reduction in a loop.

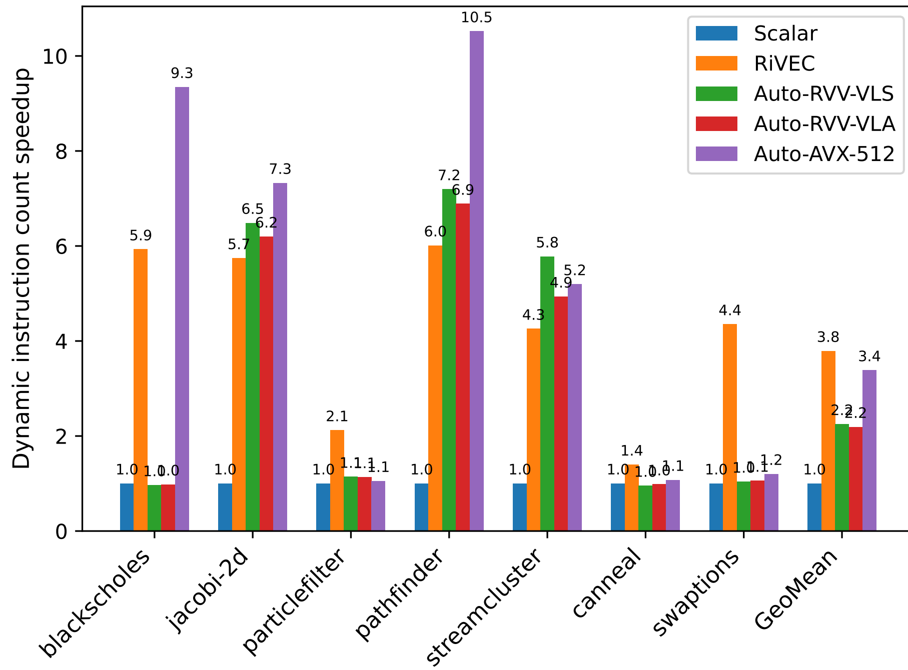
- **Reverse loop traversal**: Vector memory requests need register reversal but the shuffling cost is undefined for VLA RISC-V backend.

  Define the reversal cost for RISC-V backend.

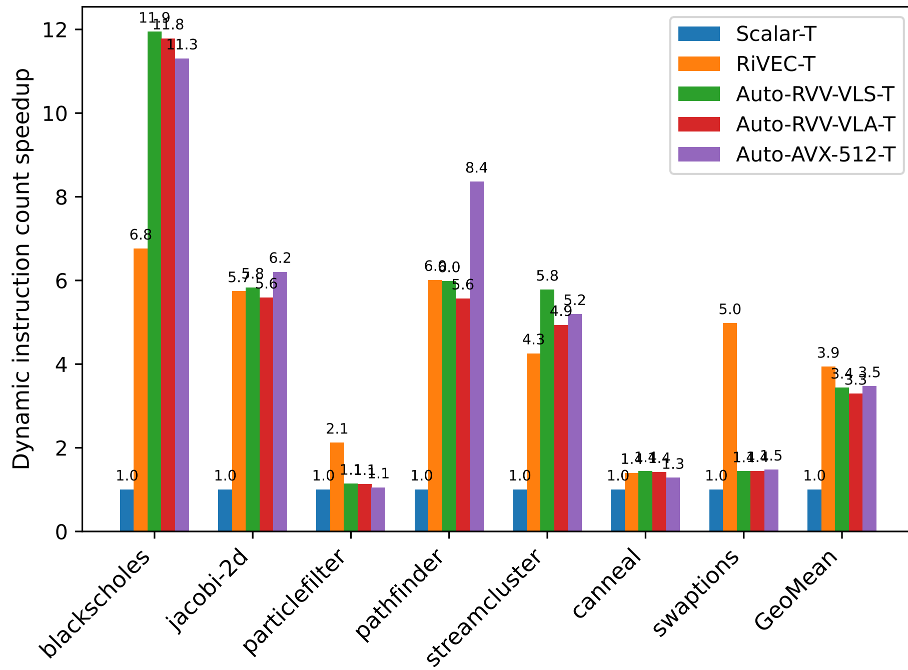
2.5 Application Benchmark Study

To complement the synthetic loop study in the previous section, we also measure real benchmarks. For this study, we need a benchmark suite with existing hand-vectorized implementations for RISC-V. As far as we are aware, only one such suite exists: RiVec [91]. We extend the benchmark suite to work with the upstream LLVM repository, which now supports RVV v1.0.

We begin by comparing the performance of the hand-vectorized and auto-



(a) Unmodified benchmarks.



(b) Transformed benchmarks (labeled as $-T$)

Figure 2.2: Dynamic instruction count speedup over the scalar version using a vector length of 8. Transformations for the auto-vector configurations are based on Table 2.2, whereas the serial and RiVec versions are transformed to skip math functions only.

vectorized versions of the benchmarks, unmodified. This initial measurement reflects the performance gap on a current LLVM with no programmer cooperation whatsoever. To understand the makeup of this gap, we then conduct a series of experiments to measure the influence of compiler optimizations, programmer effort, or changes in the programming model. Each experiment carefully modifies the auto-vectorized source code in a specific way to approximate the impact of a potential change. Table 2.2 lists the modifications for each application. We use these experiments to quantify the potential impact of an improvement in compilation or programming for vector ISAs.

2.5.1 Unmodified code

Figure 2.2a compares the dynamic instruction count speedup over scalar code on corresponding ISAs (RVV or AVX-512), of the hand-vectorized and compiler-generated configurations at a hardware vector length of 8.

Streamcluster: The compiler can effectively auto-vectorize the critical function: `dist` which has a streaming regular access pattern and a reduction operation. Figure 2.2a shows that the compiler auto-vectorized configurations have an even lower instruction overhead (better speedup!) than the hand-vector counterpart. The hand-vectorized configuration uses vector control instructions within the loop for dynamic vector length scalability (discussed in detail later) which increases the overall instruction overhead.

Blackscholes: It is embarrassingly parallel but the RISC-V auto-vectorized versions (RVV-VLA and RVV-VLS) have no speedup over the scalar version. The compiler is unable to vectorize math function calls rendering a scalar code

for the RVV-VLA configuration. In the RVV-VLS configuration, the compiler serially unrolls math function calls to process them on scalar machine before switching back to vector computation resulting in expensive register spilling and high instruction overhead. However, the compiler can use GLIBC vector math library for AVX-512 which results in a 9.3× speedup over the scalar version.

Jacobi-2d,Pathfinder: All the auto-vectorized configurations vectorize the applications to get comparable speedup to hand-vectorized version. However, the auto-vector configurations fail to identify data reuse patterns leading to redundant memory accesses.

Particlefilter,Swaptions: For these benchmarks, the compiler is unable to auto-vectorize critical sections resulting in minimal speedup over scalar code.

Table 2.1(B) summarizes the areas of improvements needed to improve performance of the compiler auto-vectorized code when compared to the hand-vector version. We discuss these gaps in detail using the context of the evaluation results.

2.5.2 Vector math libraries

For some benchmarks, we find that a significant impediment to compiler auto-vectorization in RISC-V is the use of math function (`libm`) calls in otherwise vectorizable code. An inner loop may be easily parallelizable but contain a call to a scalar `log10`, for example, that prevents LLVM from vectorizing the entire loop. Both Blackscholes and Swaptions have such function calls in the critical

sections of the code.

To measure the performance impact of this limitation, we construct special versions of the two affected benchmarks that "factor out" the influence of these math functions. In both the hand-vectorized and auto-vectorized versions, we replace the problematic math functions with no-ops. The resulting comparison approximates the remaining performance gap *if the compiler could perfectly auto-vectorize code with math functions*.

Figure 2.2b shows the results all transformed benchmarks: in Blackscholes, factoring out math functions closes the gap entirely, but auto-vectorization for Swaptions is still limited by other factors (discussed ahead). The auto-vectorized configurations after transforming Blackscholes, have over 11× speedup compared to the 6.8× speedup for the hand-vector counterpart. This margin is due to better fused-instruction selection and loop invariant optimization by the compiler. However, since math lib calls take a major fraction of the code execution, the hand-vectorized version might have glossed over these optimizations.

These advantages show that the compiler does a good job at instruction selection and optimizations, for simple compute patterns. Moreover, auto-vectorization can take advantage of the boring, fiddly optimizations and let programmers focus on the bigger picture.

> LLVM should support auto-vectorizing code with `libm` calls by replacing them with calls to a vectorized math library for RISC-V.

2.5.3 Vector-scalar width mismatch

The RISC-V vector extension (RVV) and AVX ISA allows a flexible element width in vector registers, in contrast to the fixed-width scalar registers defined by the base RISC-V ISA. This flexibility can cause problems when code has interactions between scalar and vector values. If an application uses 32-bit values everywhere but is compiled for RV64, then the scalar values will be promoted to 64-bit registers (using the `i64` type in LLVM). The values in vector registers, however, remain 32-bit values (e.g., using the `<8 x i32>` vector type in LLVM).

The result is that the compiler generates unnecessary instructions to convert between different element widths and might use extra vector registers to accommodate widened elements. To avoid this pitfall, we fix the primary data type for all benchmarks to use 64-bit values and compile for the RV64 base ISA. However, to make auto-vectorization more accessible, LLVM and other compilers should evolve to elegantly handle element size mismatches.

> LLVM should infer scalar width from vectorized data types.

2.5.4 Dynamic vector length scalability

When programming with RVV intrinsics, programmers can stripmine loops and dynamically adjust the number of elements handled per iteration:

```
//dynamic vector length
int hwl = vsetvl(N);
for (int i = 0; i < N; i += hwl){
  hwl = vsetvl(N-i);
```

```
  ...
}
```

This adjustment is especially useful in cases where the loop trip count is not a multiple of the maximum hardware vector length. However, the LLVM auto-vectorization only executes vector code in the maximum hardware vector-width, shown using pseudo-code:

```
//maximum hardware vector length
int max_hwl = read_csr_vlen();
for (int i = 0; i < N; i += max_hwl){
  if ((N-i)<max_hwl) break; //execute left-over as scalar code
  ...
}
```

This can lead to poor scalability, usually with larger vector units. Figure 2.3 shows poor scalability in Jacobi-2d for the compiler generated auto-vectorized versions since the loop trip count is not perfectly divisible by vector length (due to a convolution-style computation). We notice no apparent instruction reduction for both VLA and VLS auto-vectorized versions on going from vector length 16 to 32 since the scalar overhead of loop "tails" offsets the instruction savings from the increased vector length.

LLVM allows predication based vectorizing of the loop tail using the flag: `-prefer-predicate-over-epilogue=predicate-else-scalar-epilogue` but this is orthogonal to dynamic vector length control in RVV and can cause unnecessary register spilling in larger loops with conditional branches.

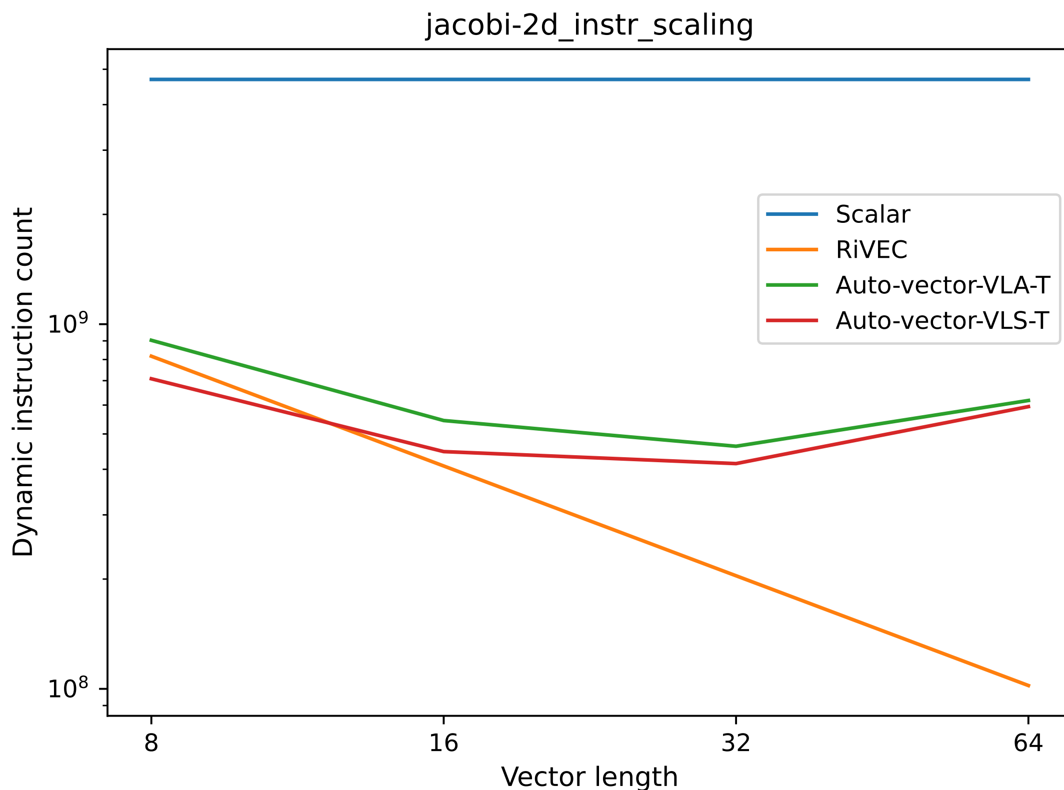


Figure 2.3: Dynamic instruction scaling across different hardware vector length in Jacobi-2d plotted on logscale. The overhead of running scalar instructions (due to non-scalability) increases at higher vector length for compiler generated code.

> LLVM should generate loops that embrace the scalable vector style: instead of assuming a fixed vector length and using scalar instructions for loop "tails," it should generate code that uses `vsetvl` to dynamically adjust the length on every iteration.

2.5.5 Shuffle pattern detection

Both Pathfinder and Jacobi-2d have overlapping memory access patterns, which is illustrated using a simplified example below:

```
for (int i = 1; i < N; i++) {
  b[i] = a[i-1] + a[i] + a[i+1];
}
```

The hand-vectorized code uses RVV shuffle instructions: `vslide1up` for `a[i-1]` and `vslide1down` for `a[i+1]` to shift the values in the vector register of `a[i]`, avoiding redundant memory accesses. Such an optimization entails two components for the compiler:

- Analyzing overlapping memory access patterns to remove redundant loads.
- Representing shuffle patterns in the IR and selecting optimal instructions in the backend.

In general, selecting special-purpose vector shuffle instructions is hard for compilers [112]. LLVM can analyze simple recurrence patterns in the absence of aliasing but fails in more complicated cases like conditional branches (in Pathfinder) and 2D array accesses (in Jacobi-2d). We apply transformations from Table 2.2 to simplify these applications to the example discussed above and assess LLVM performance. The manual transformations allows the compiler to recognize one (out of the two) first-order recurrence pattern between `a[i]` and `a[i-1]`.

LLVM uses a *dedicated* IR intrisic (`llvm.experimental.vector.splice`) for representing this pattern for VLA configuration, unlike the more standard `shufflevector` instruction used for VLS. These instructions are lowered to RISC-V shuffle instructions: `vslidedown` and `vslideup`, in the backend.

Figure 2.4 shows the decrease in memory requests in the transformed auto-

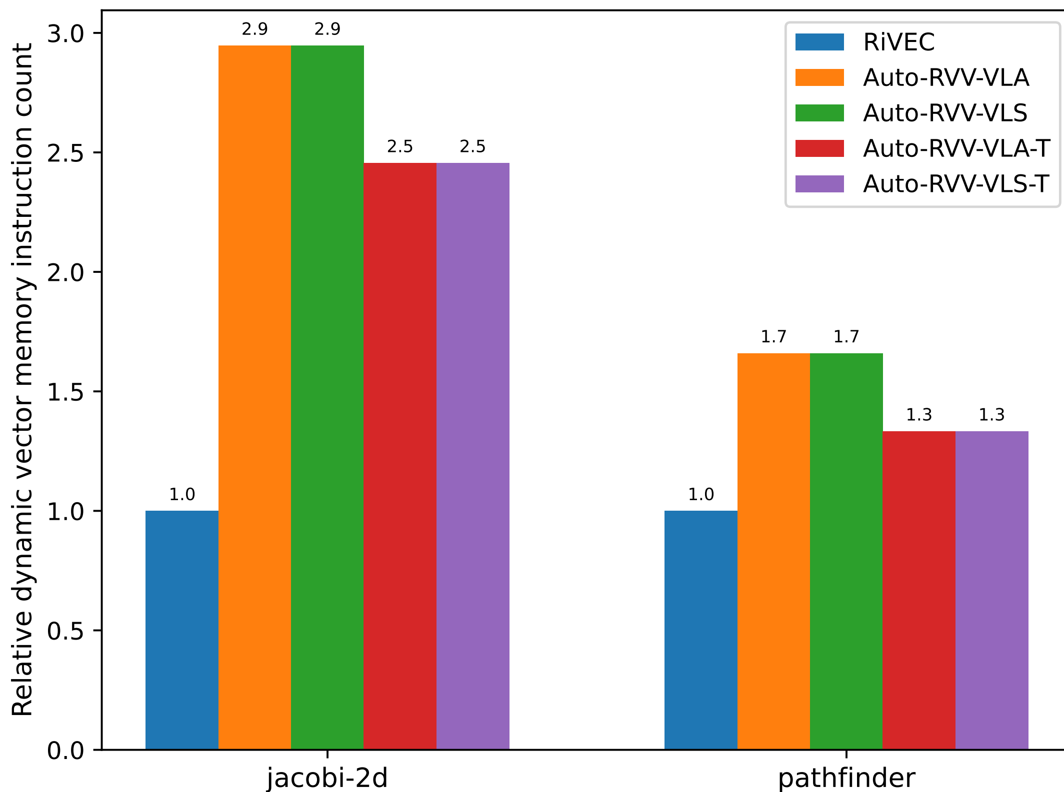


Figure 2.4: Vector memory requests relative to hand-vectorized baseline. The transformation allows compiler to reduce redundant loads in both the auto-vectorized configurations.

vector configurations for both benchmarks. Since the compiler is partially successful in reducing redundant memory loads, the transformed auto-vector configurations still produces 2.5× and 1.3× higher memory requests compared to the hand-vectorized version for Jacobi-2d and Pathfinder respectively.

In some cases, the shuffling patterns across vector elements form the core of critical loops. Particlefilter is one such case, where the compute pattern is expressed using a sophisticated instruction `vfirst`, designed to select the first nonzero vector element. Hence, the compiler fails to vectorize the critical sections of the benchmark leading to poor performance.

LLVM needs to improve shuffle pattern analysis for generic and backend specific patterns and use generalizable mask representation for VLA configuration.

2.5.6 Algorithm driven Loop Fusion

A lot of intrinsic-based vector programming comes down to customizing algorithms to achieve better performance for a given configuration.

The hand-vector version of Canneal uses loop fusion, among other techniques, to improve vectorization. A simplified code block from Canneal is shown below:

```
for (int i = 0; i < fanin; ++i){
  a = a + fanin_val[i];
}
for (int i = 0; i < fanout; ++i){
  a = a + fanout_val[i];
}
```

Since the loops are restricted by graph `fan-in` and `fan-out` degrees, even at large data simulations, the loop bounds can be smaller than the hardware vector length. In such cases, fusing the loops can provide efficient vectorization opportunities to scale to larger hardware vector lengths. However, loop fusion is not trivial and requires setting up combined arrays to facilitate it. We perform this transformation, inspired from the hand-vectorized version, for the compiler auto-vectorization configurations:

```
for (int i = 0; i< fanin+fanout; ++i){
```

```
  a = a + all_val[i]; // has both fanin, fanout nodes
}
```

This algorithmic transformation allows auto-vectorized code to run vector instructions at the maximum supported hardware vector length and close the gap with hand-vectorized version as shown in Figure 2.2b.

> Future programming model for vectorization should be able to guide programmers towards transformations such as Loop Fusion.

2.5.7 Vectorizing specific loops

In general, LLVM's auto-vectorization focuses on vectorizing the innermost loop in each loop nest. In situations where interchanging loops is not trivial, the compiler might fail to see vectorization opportunities or vectorize irrelevant loops. This effect arises at various places in Swaptions. One such instance, after interchanging loops and simplifying 2D accesses, looks like this:

```
for (int i = 0; i < N; ++i){
  int sum = 0;
  for(int j = 0; j < M; ++j){
    sum += a[j][i];
  }
  b[i] = c[i] + sum;
}
```

In the benchmark: $M = 3$, so just vectorizing the inner loop is not very useful. In addition, even if the inner loop were not vectorizable, the compiler would give up and not look at broader vectorization opportunities that might be visible

to the programmer. The hand-vectorized version can vectorize the outer loop, which is much more scalable due to the larger loop trip count (known to the programmer) and results in simpler unit-strided vector memory accesses. This strategy allows the hand-vector version to scale well with increasing hardware vector length. The auto-vectorized versions fall short due to focusing on innermost loops by default.

> LLVM auto-vectorizer should support outer-loop vectorization. However, identifying scalable loops for vectorizing requires simultaneous loop interchanging optimizations and cost analysis, which can be hard for compilers and should be offloaded to programmers.

2.5.8 Adapt algorithms to the microarchitecture

In the previous code example, the variable N is used as a blocking parameter for better caching in the serial version of the code. The hand-vector code changes the algorithm to set the variable to the hardware vector length using vector intrinsics. This unique feature of vector-length agnostic ISAs like RVV allows the algorithm to automatically adapt to different microarchitectures.

While high-level algorithmic changes are out of scope for a traditional C compiler, they represent an opportunity for higher-level languages and DSLs that compile to vector ISAs [89].

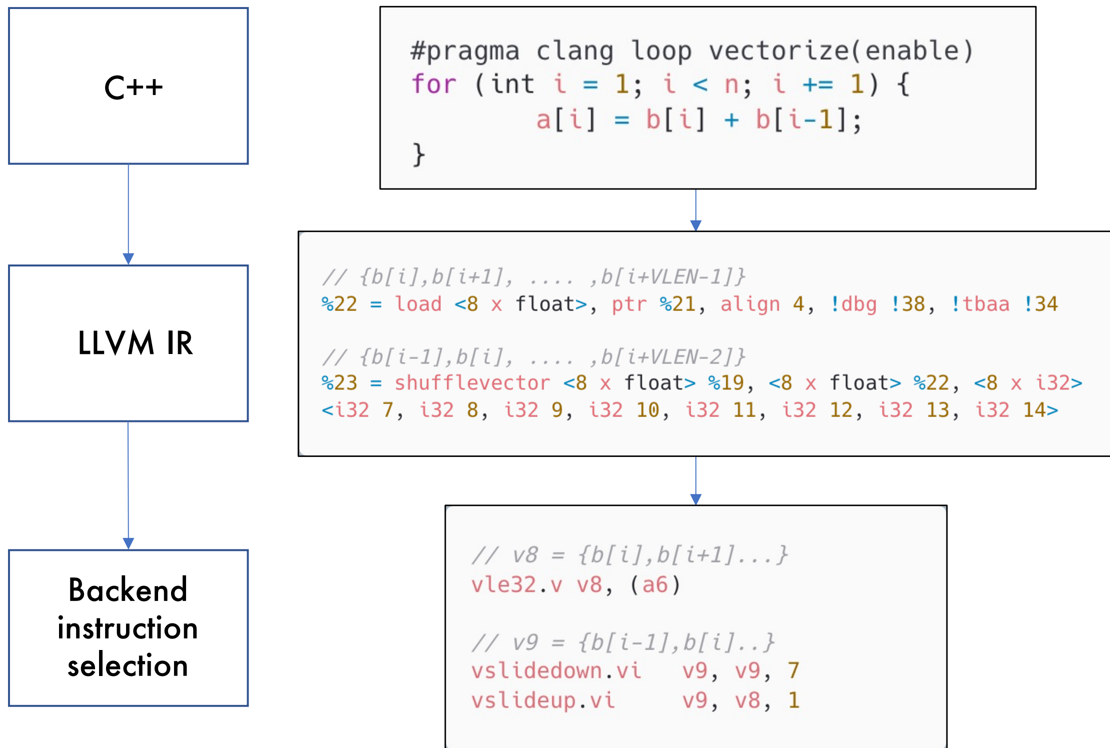


Figure 2.5: Instruction selection procedure in LLVM vectorization for fixed vector-length configuration of RVV. The LLVM IR represents the data movement across vector registers using the `shufflevector` intrinsic using a fixed vector-length mask array. This representation fails for length-agnostic designs.

> Future work should explore programming models that make this kind of algorithmic parameterization available to programmers without requiring manual tuning of hardware intrinsics.

2.6 Solution proposal: Scalable compiler IR

In this section, we discuss a possible solution that can address some of the issues pointed out in the previous sections.

The compiler IR is a target-independent respresentation to decouple com-

piler frontend optimizations from the backend device. Figure 2.5 shows the LLVM IR representation for `shufflevector` intrinsic and its backend specific lowering to ISA instructions. First, the vectorization pass identifies a first-order recurrence pattern in the loop. The IR uses vector register indexes to represent data shuffling. In the RISC-V backend, the offsets can be pattern matched to the optimal instruction.

This procedure worked well with traditional fixed vector-length ISAs, but does not extend directly to length-agnostic designs since vector-length is unknown at compile time. Currently, LLVM [57] uses a *dedicated* IR intrisic—`llvm.experimental.vector.splice`(inspired from the ARM SVE ISA) for length-agnostic ISAs to represent this pattern. This breaks the decoupling between the frontend and backend of the compiler. Every new instruction requires coordinating 3 changes across 3 layers: changing the auto-vectorizer, changing the IR and changing the backend. This is in contrast to the way compilers are *supposed* to work: to add a new instruction you only modify the backend. We want to get back to this place.

While LLVM uses dedicated intrinsics to represent the register shuffle patterns, it uses a unified intrinsic—`llvm.masked.gather` and `llvm.masked.scatter` for memory access patterns that are not contiguous accesses. However, the offset pointer for gather/scatter intrinsics are represented using fixed-length arrays for VLS ISAs, similar to shuffle masks. These arrays can be pattern-matched to backend specific instructions such as "strided load" in VLS configuration but breaks down for VLA designs.

To solve the representation issue, we propose designing the ScaleIR. We make the following modifications in the compiler for a prototype design:

```
// vscale = VLEN
%vscale = call i32 @llvm.vscale.i32()
// stepvec = {0, 1, 2, ... , VLEN-1}
%stepvec = call <vscale x 2 x i32> @llvm.experimental.stepvector.nxv2i32()

// {..}
// %11 = f(vscale,stepvec)        Arithmetic function

%22 = call <vscale x 2 x i32>
@llvm.experimental.vector.shuffle.nxv2i32.nxv2i32(<vscale x 2 x i32>
%vector.recur, <vscale x 2 x i32> %wide.load20, <vscale x 2 x i32> %11)
                                                    Shuffle mask
```

Figure 2.6: Represent shuffle masks using a function of vector-id and scalable vector length in the IR.

- Modify the loop vectorization pass to represent shuffle and gather/scatter masks using arithmetic functions of vector lane-id (`llvm.experimental.stepvector`) and scalable vector length (`llvm.vscale`), as shown in Figure 2.6.
- In the RISC-V backend, traverse the IR Control Flow Graph (CFG) to compose arithmetic functions to identify patterns for optimal instruction selection.

We perform a preliminary evaluation of this prototype design for ScaleIR on the TSVC benchmark loops. We found the following auto-vectorization improvements using ScaleIR:

- ScaleIR bridges the instruction selection gap discussed in Section 2.4 by selecting strided loads in all relevant loops for the VLA configuration. This is enabled by the IR modification that represents the strided access pattern as a function of stride offset and loop iteration variable, and the backend pass to detect and select the RVV specific instructions: `vlse`,`vsse`.

- We extend RVV backend passes to select optimal shuffle instructions such as `vslide1up`,`vslide1down` for relevant patterns.
- ScaleIR is not limited to `llvm.experimental.vector.splice` intrinsic with constant offset argument. This flexibility allows it to support shuffling patterns with variable vector-length offsets. Hence, we are able to support product reductions using tree-based vector register reduction in a dynamic loop (proposed in Table 2.1-A4).

Currently, the instruction selection for shuffle patterns is limited by the ones supported in the loop vectorization pass. LLVM extends scalar optimizations such as first-order loop recurrence, to specific vector shuffle instructions like `splice` with a constant offset of 1. Ideally, LLVM should analyze a wide range of register shuffle patterns (similar to main-memory optimizations such as coalescing interleaved memory accesses [71]) and represent using a `shufflevector` mask pattern. This general mask pattern opens opportunities for the backend to select ISA specific instructions based on patterns expressed in the IR. ScaleIR also enables a unified approach to represent VLS and VLA configurations by taking a vector-length agnostic approach. In the future, ScaleIR could enable the following directions of work:

- Decouple shuffle pattern analysis in the Loop Vectorization pass by representing a general pattern in the IR and selecting specific instructions in backend. Finding *generic* recurrences that are insensitive to vector lengths can increase vectorization opportunities. This would also involve a different approach for cost modelling since the backend decides the feasibility of the shuffle instruction and rolls back the shuffle if not effective.
- Automatically generate backend passes to detect ISA specific patterns. We

believe there should be a DSL to generate these passes based on an arithmetic function of the pattern. The DSL could also enable correctness verification of the passes, which can get complex due to graph traversal and pattern matching.

This work would extend the shuffle pattern detection in loop vectorization (proposed in Table 2.1-B4), in context of length-agnostic ISAs.

2.7 Conclusion

Next-generation vector ISAs portend a new era for mainstream parallel programming models. Their popular uptake, however, requires moving beyond manual intrinsic-based programming. The goal should be to let programmers express high-level parallelism strategies while letting the compiler focus on what compilers do well: selecting instructions, scheduling computations, and removing redundancy. To this end, compiler designs should incorporate suggestions outlined in this chapter to expand support for emerging ISAs and consequently enable improved programmability and high-performance code generation.

CHAPTER 3

SOFTWARE-CONTROLLED HARDWARE PREFETCHING

3.1 Introduction

Data prefetching in modern datacenter workloads has an accuracy problem. Per-core memory bandwidth is reaching a plateau [42, 67], but datacenter workloads' bandwidth consumption is still rising about 10% each year [42]. In environments with such constrained bandwidth, prefetchers must act conservatively: they must not waste bandwidth in exchange for marginal cache hit-rate improvements. Unfortunately, hardware prefetchers such as Stride, Best Offset Prefetcher (BOP) [62] and Signature Path Prefetcher (SPP) [48], have a fundamental tradeoff between higher bandwidth consumption and improved cache hit rates (see Figure 1.1), as covering more misses invariably results in more speculative and useless prefetch requests. While this tradeoff can be justified in a bandwidth-rich environment, aggressive prefetchers are an increasingly poor fit for datacenter workloads, and recent work [42] has even shown that disabling hardware prefetchers altogether can sometimes improve datacenter performance.

Two possibilities for more accurate, conservative prefetching include *software prefetching* and *throttled hardware prefetching*. Software prefetching [17, 64, 66, 65, 58, 4, 43] achieves high accuracy by putting workloads in control of exactly what to prefetch. However, its dependence on programmer or compiler guidance can make it difficult to apply universally, which entails guessing where to insert a prefetch instruction to achieve an optimal "lead time" [54]. Software prefetching also incurs inherent instruction overheads.

Hardware throttling [107, 11] is a promising alternative: it can dynamically detect situations where a hardware prefetcher is wasting bandwidth and suppress it. Inevitably, however, hardware throttling is *reactive*—it must observe some bandwidth waste before it can correct it. Moreover, state-of-the-art hardware throttling solutions [11] require careful workload-specific tuning, so configurations that work on one set of workloads, such as SPEC, may not generalize to the diversity of workloads in the datacenter. We find (in Section 3.2) that the reactive and workload-sensitive nature of hardware throttling can cause problems in a datacenter setting, where instruction footprints are large and context switches frequent.

This paper proposes to combine the strengths of software-driven and hardware-driven approaches to improving prefetching accuracy. Like hardware throttling, our goal is to start with an aggressive prefetcher and opportunistically attenuate its bandwidth waste to achieve both high accuracy and high coverage. Like software prefetching, we aim to adapt to workload-specific data access patterns instead of requiring universal heuristics. Our approach uses *software directives* to control the aggressiveness of hardware prefetching. The result is a hardware–software collaboration: software is responsible for deciding which code can benefit from prefetching, and hardware remains responsible for predicting which data to prefetch and when. This division of labor exploits the strengths of each domain: software can "program" the hardware prefetcher based on workload requirements or system conditions, and hardware can exploit signals from dynamic program behavior to make specific prefetching decisions. In other words, software ensures accuracy and hardware maximizes coverage.

Specifically, we envision an expansion of the hardware–software interface to control prefetching. Current hardware interfaces offer software only two options: (1) instructions to unconditionally prefetch specific data, or (2) disable hardware prefetching entirely. In our proposal, software can mark code or data to enable or suppress hardware prefetching in a particular context. By allocating these markers more or less aggressively, workloads can navigate the trade-off depicted in Figure 1.1 much more effectively.

Our implementation, which we call Programmable Prefetching (ProP), is a profiling-based mechanism for generating software markers that suppress prefetching when it likely to be inaccurate, but the same interface can also support similar hints from static analysis or programmer annotations. We show how to generate these markers, how to cheaply communicate this information from software to hardware without requiring ISA changes, and how the hardware prefetcher can leverage these markers to filter prefetches.

Overall, we make the following contributions:

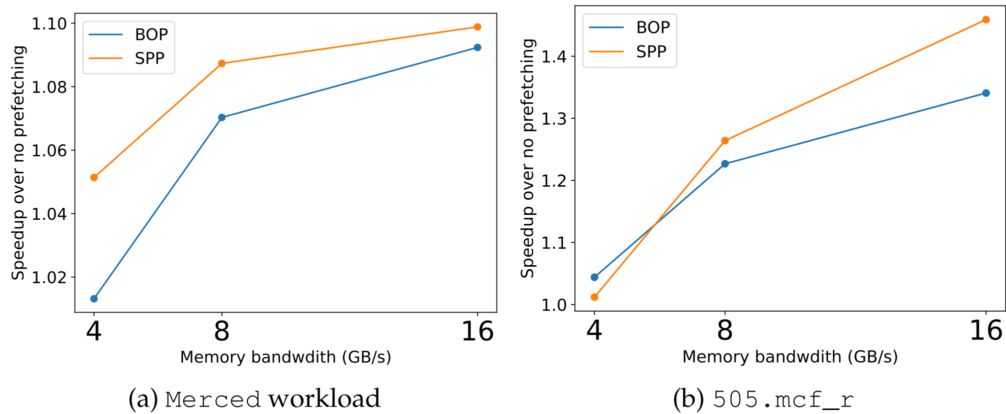
- We show that state-of-the-art prefetch filtering schemes do not work well in datacenter environments. In particular, we show that bandwidth constraints, frequent context switches, and large workload diversity makes hardware throttling ineffective in datacenters.
- We propose a new hardware-software interface to modulate prefetchers effectively, and we show that our proposed division of responsibilities between hardware and software can provide superior tradeoffs than hardware or software prefetching alone.
- We design, implement, and evaluate ProP, a profile-guided tool that associates prefetch directives for PCs.

- We evaluate ProP on multiple state-of-the-art prefetchers and across datacenter workloads and SPEC 2017 INT workloads using the gem5 simulator. On average, ProP increases performance by 3% over the state-of-the-art underlying prefetcher, SPP, and 1.8% over a hardware-based prefetch filtering technique, PPF. With frequent context switches, the benefit of ProP increases performance to 4% over the hardware baselines.

3.2 Motivation

Prefetchers improve performance by predicting future memory accesses and fetching them ahead of time in the cache. There are two prefetcher metrics that measure how good the prefetcher prediction is: *accuracy* and *coverage*. Prefetcher accuracy is the fraction of prefetches that were useful and subsequently used by a demand access. Prefetcher coverage is the reduction in demand misses due to the prefetching.

In bandwidth-rich environments, prefetcher coverage is the primary indicator of its performance benefit: the higher the coverage, the better the prefetcher performs. However, in bandwidth-constrained environment—common in datacenter environments—the performance tradeoffs are more complex. In bandwidth-constrained environments, while high coverage can still help hide memory latency, low accuracy can degrade performance by worsening bandwidth contention. Figure 3.1 confirms this as we see that the performance improvements for two state-of-the-art prefetchers, Signature Path Prefetcher (SPP) [48] and Best Offset Prefetcher (BOP) [62], get narrower as bandwidth becomes scarce.



(a) Merced workload (b) 505.mcf_r

Figure 3.1: The benefit of hardware prefetchers shrink with limited bandwidth on both Merced (a large datacenter workload) and mcf (SPEC workload).

There are two prevailing strategies for tackling the performance loss in bandwidth-constrained settings. The first strategy is reactive [107, 78]: it dials the prefetcher back when bandwidth contention is observed. The second strategy is predictive [49, 11]: it learns at a fine-granularity which prefetch requests to drop.

3.2.1 Reactive Throttling

Reactive solutions observe bandwidth waste and throttle the prefetcher using run-time information [107, 78]. For example, for SPP [48], such throttling can be achieved by changing the confidence threshold to modulate prefetcher aggressiveness, for BOP [62], it can be achieved by changing the offset score thresholds used to calculate the best offset, and for IP-stride, aggressiveness can be modulated by changing the degree and look ahead distance of the prefetcher. Figure 3.2 shows that for all three prefetchers, throttling inevitably has a sharp accuracy–coverage tradeoff, where a lot of coverage is sacrificed

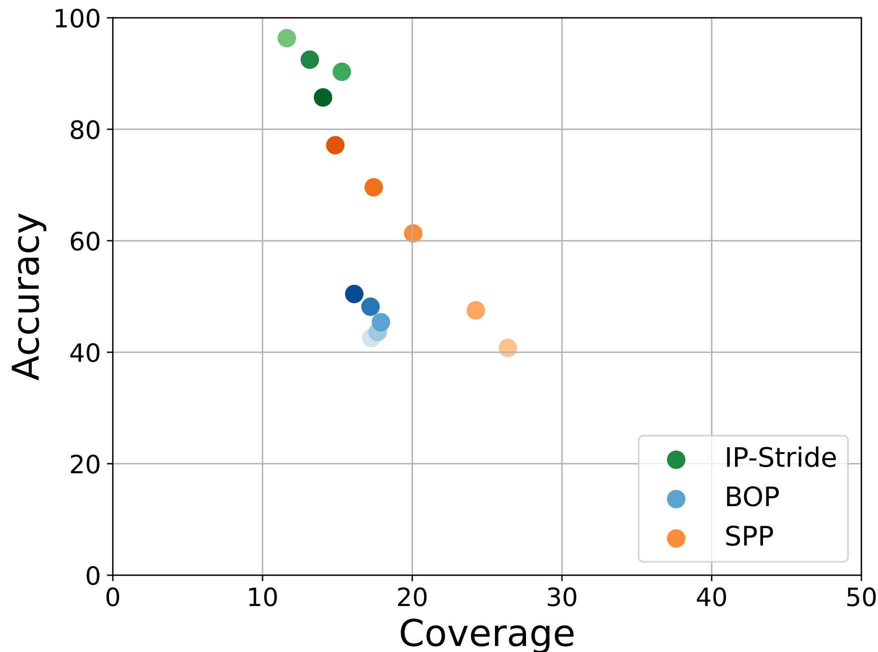
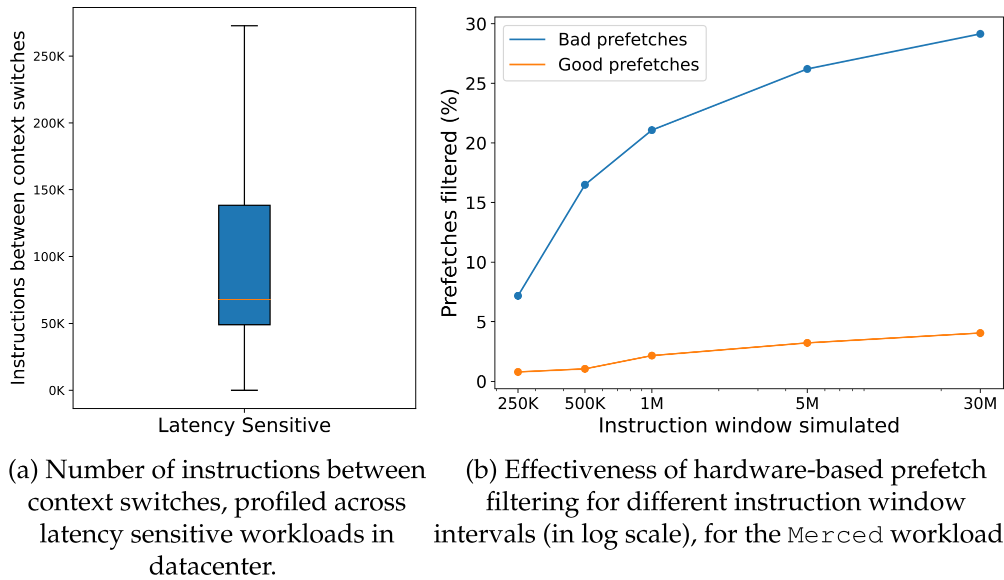


Figure 3.2: Accuracy vs coverage tradeoff for different configurations of IP-stride, Best Offset (BOP) and Signature Path (SPP) prefetchers, on `Merced` and `Bravo` datacenter workloads.

to get marginally better accuracy. This is because throttling is very coarse-grained, where prefetching is suppressed uniformly, meaning that "productive" prefetches are supressed just as much as "unproductive" ones, without discriminating between them.

3.2.2 Predictive Throttling

Predictive solutions [49, 11] learn to drop prefetch requests more selectively instead of throttling *all* prefetch requests. For example, the perceptron prefetch filter (PPF) [11] uses multiple program features and a perceptron learning algorithm to identify prefetch requests that should be dropped. Unfortunately, these solutions are not effective in datacenter environments for several reasons:



(a) Number of instructions between context switches, profiled across latency sensitive workloads in datacenter.

(b) Effectiveness of hardware-based prefetch filtering for different instruction window intervals (in log scale), for the `Merced` workload.

Figure 3.3: Context switches are frequent in datacenter workloads, making it difficult for online throttling schemes to warm up adequately.

Frequent context switches Learning prefetcher effectiveness dynamically at runtime requires time to learn and warmup that is difficult in datacenter environments, where applications are highly-multithreaded and incur frequent context switches (Figure 3.3a). Figure 3.3b shows PPF's ability to filter bad prefetches issued from the underlying prefetcher - SPP, is diminished at smaller instruction windows.

Workload diversity Filtering techniques like PPF (and often the underlying prefetchers like SPP) have a wide range of parameters that require workload-specific tuning for the prefetcher to work effectively. A typical datacenter runs thousands of workloads, which are constantly evolving. Hard-coding hardware prefetcher values at design time based on a few workloads is unlikely to yield effective results in the long life of a server platform.

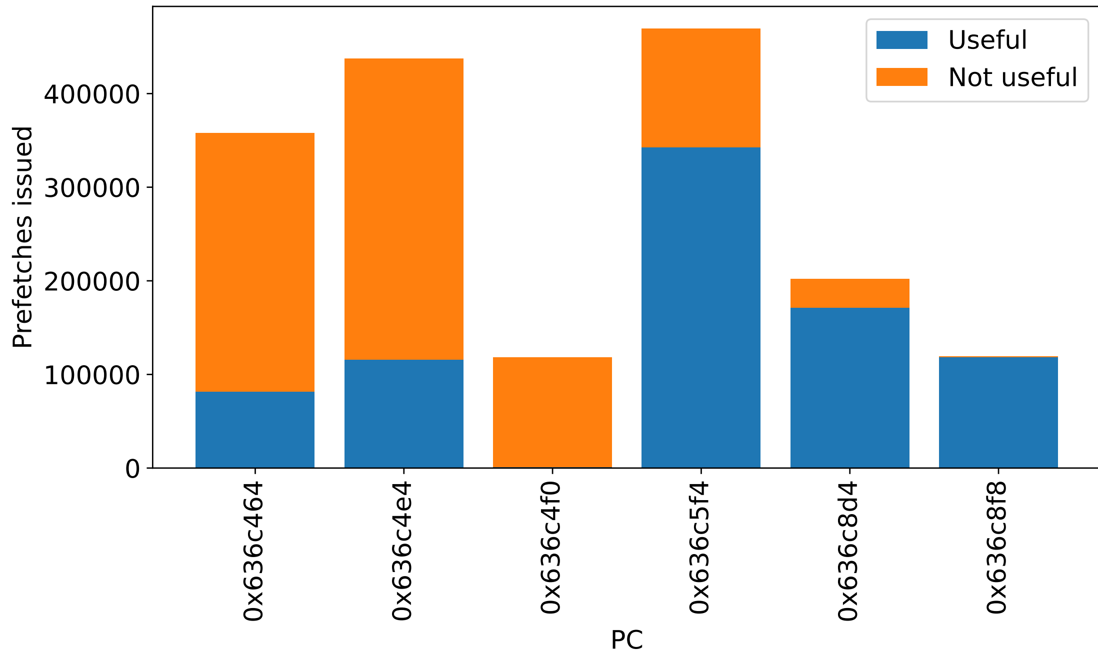


Figure 3.4: Prefetching accuracies with BOP for each PC in a code region.

Large instruction footprint Finally, tracking features such as program counters and page addresses within limited hardware budgets is infeasible for datacenter workloads because they have large instruction and data footprints. For example, our evaluated datacenter workloads have more than 4× the instruction footprint of perlbench—the largest among SPEC workloads, and this footprint is constantly growing [46]. Using hardware resources to track prefetching behavior at a fine granularity can quickly require ~100s of KBs of hardware resources for tracking.

3.3 Profiling Insights

BOP is a region prefetcher based on the Sandbox prefetcher [88], which learns a prefetch offset from all memory accesses in a training window. Since it does

not rely on tracking individual load/store streams, it can be cost-effective for large PC-footprint workloads and finds inter-stream prefetching opportunties as well. However, once the offset is determined, prefetches are issued for all memory accesses. This can lead to large traffic overheads when non-streaming accesses are interleaved in the region, which are not prefetch friendly. Figure 3.4 shows the difference in prefetch accuracies issued by BOP, visualized for each PC in a selected region of high prefetching for a datacenter workload. Unfortunately, even though some PCs are almost always inaccurate for prefetching, they get issued by the prefetcher since the entire region was adjudged as prefetch friendly.

We simulated several threads across datacenter workloads and observed code and data patterns that determine prefetching behavior. In this section, we discuss observations based on `Merced`, a large Warehouse Scale Computer (WSC) application, but these insights apply to other workloads as well.

3.3.1 Program counter maps to distinct prefetching behavior

We simulated 40 threads from `Merced`, and measured the prefetching accuracy for each PC on which the underlying prefetcher - SPP, issued a prefetch. Figure 3.5 categorizes each PC into 3 prefetch accuracy buckets. The average prefetcher accuracy across our simulation is 48%. We observe that a significant portion of those prefetches($\sim$ 40%) are issued on memory instructions that lead to poor accuracy($<$ 30%). Since we aggregate prefetching statistics across different thread executions, this is evident that prefetching behavior for these PCs is independent of the input data across different thread contexts.

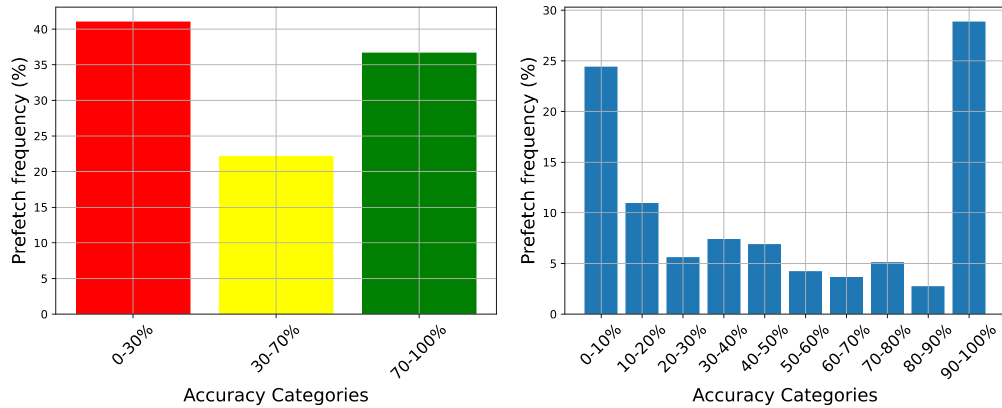


Figure 3.5: PC-based SPP prefetcher accuracy histogram, for different accuracy bin sizes. A significant portion of prefetching is done for PCs that are < 30% accurate. PCs themselves, can be a good indicator for prefetching behavior since frequencies of < 10% and > 90% bins are highest.

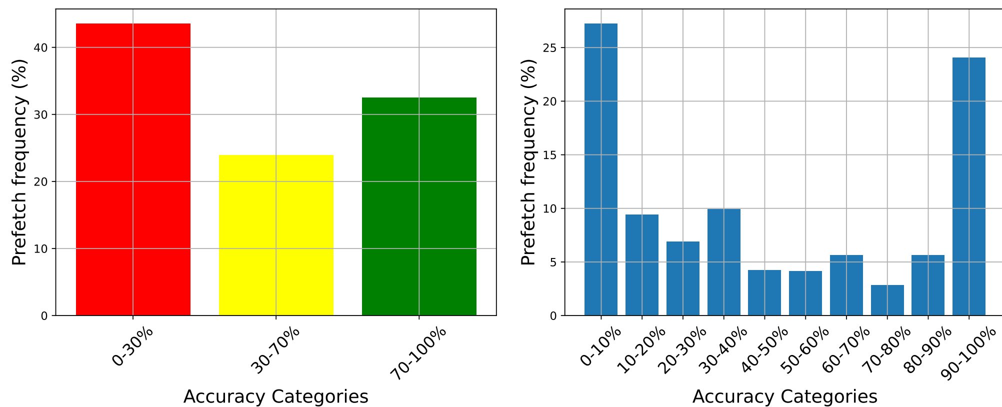


Figure 3.6: PC-based BOP prefetcher accuracy histogram, for different accuracy bin sizes.

Figure 3.6 shows that this PC-based behavior is similar across other prefetchers such as BOP, which has similar fractions of PCs in each accuracy bucket, further signaling the need of code-level features to disambiguate prefetching behavior.

To understand the distribution of these PCs. we look at aggregated prefetching behavior for higher level libraries in `Merced`'s trace execution, shown in Fig-

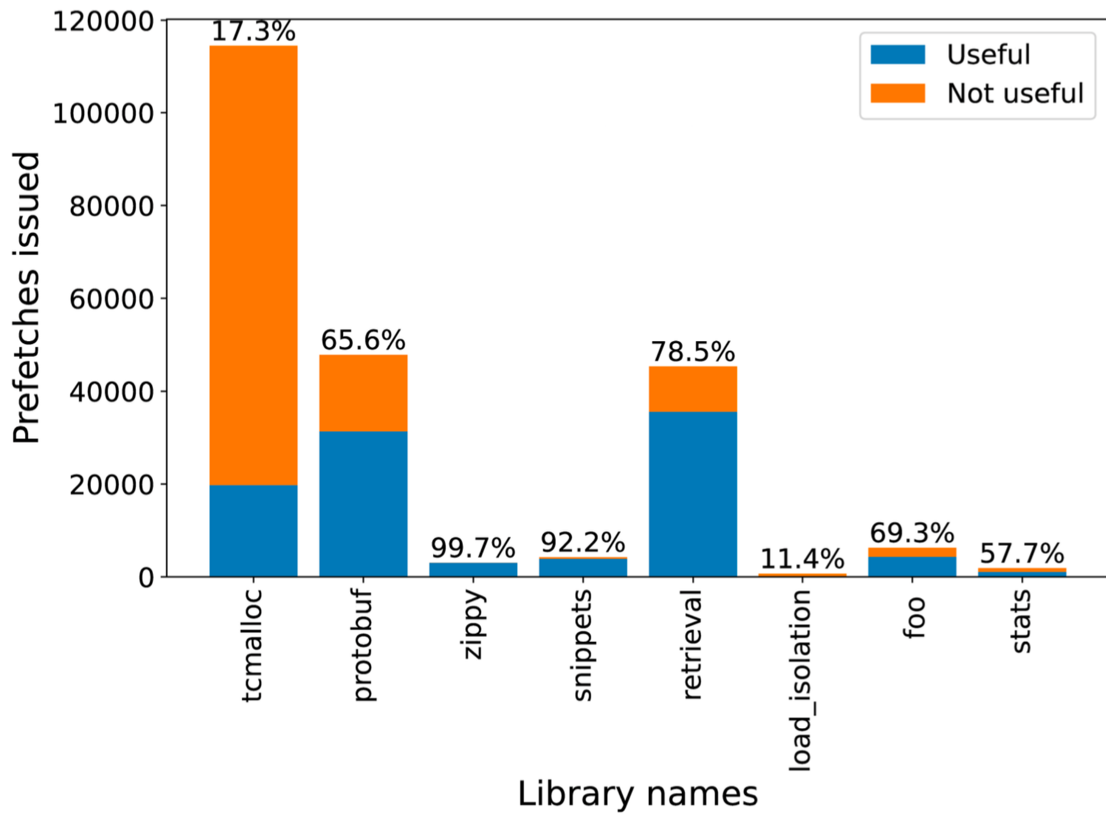
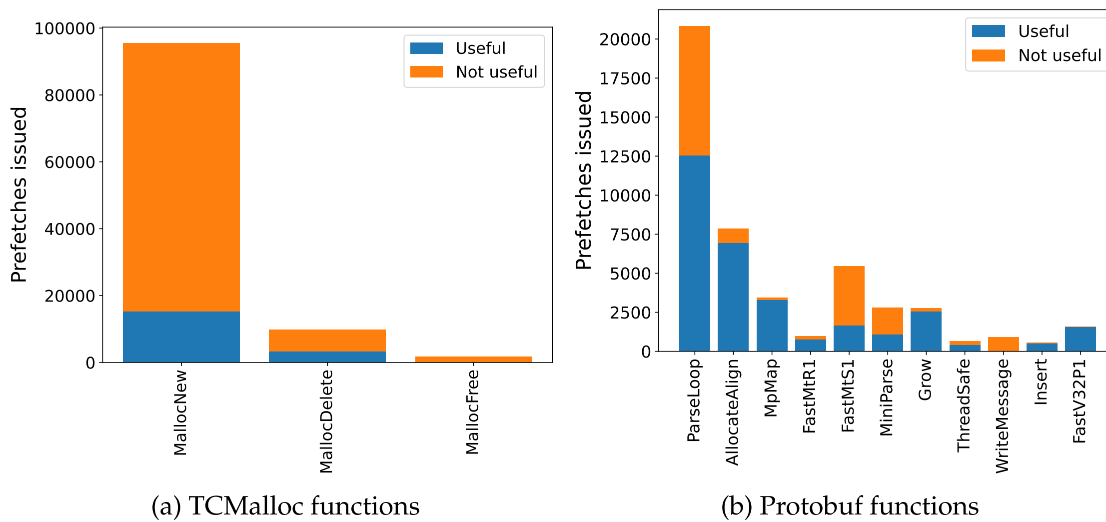


Figure 3.7: Aggregated SPP prefetcher accuracies for various libraries called during the `Merced` trace execution. `foo` is an anonymized library name.



(a) TCMalloc functions (b) Protobuf functions

Figure 3.8: Function-level breakdown of prefetching accuracies

ure 3.7. A few libraries such as Zippy, which is a compression/decompression library, have a high prefetch accuracy since the accesses are sequentially long stream with strided accesses.

On the other hand, TCMalloc library has a poor prefetching accuracy of ~17%. Figure 3.8a shows the breakdown of function accuracies within TCMalloc. The `MallocNew` function allocates a memory region from a list of available locations:

```
void* Allocate(size_t size_class) {
  return freelist_.Pop(size_class);
}
```

Due to memory fragmentation and dynamic memory allocation/deallocation, these locations in the `freelist_` are stack like, and it's direction change is random, making it hard to prefetch.

We also looked at finer-granularity of function within libraries such as protobufs, where the prefetch accuracy is not too high or low. Figure 3.8b shows that the most called function in protobufs is `ParseLoop`, which iterates over a given proto structure. Since different proto structures will have different memory layouts, this function's prefetching behavior can vary depending on the call context or data layout, which results in a 50-60% accuracy on average.

3.3.2 Code context and data features can enhance prefetching understanding

Figure 3.5 and 3.6 show that ~20-25% of prefetching behavior in `Merced` is not distinguishable by PC since the prefetch accuracy is between 30-70%. We look at additional features which can help our understanding of their memory access patterns.

Call context

As discussed in the previous section, `ParseLoop` function in protobuf, is called on different proto structures. These proto structures are statically generated and typically have nested repeated structures that can determine the memory access pattern. We collected callstack data for each PC consisting of a tree of function calls, to differentiate such cases, where two different proto structures (with different call stacks) can call the same function. For instance, for the highest prefetch issuing instruction within `ParseLoop`, 30% of the callstacks had a high prefetch accuracy (> 90%) and 10% had an accuracy < 10%. The highly accurate call stacks come from Stream or Batch based protos, which are sequential strided patterns.

The datacenter workload traces have PGO and cross-module thinLTO optimizations that enable deep callsite inlining behavior [6]. This inlining embeds partial call context data in the instruction pointer itself. However, as discussed above, additional call context information can further differentiate prefetching behavior.

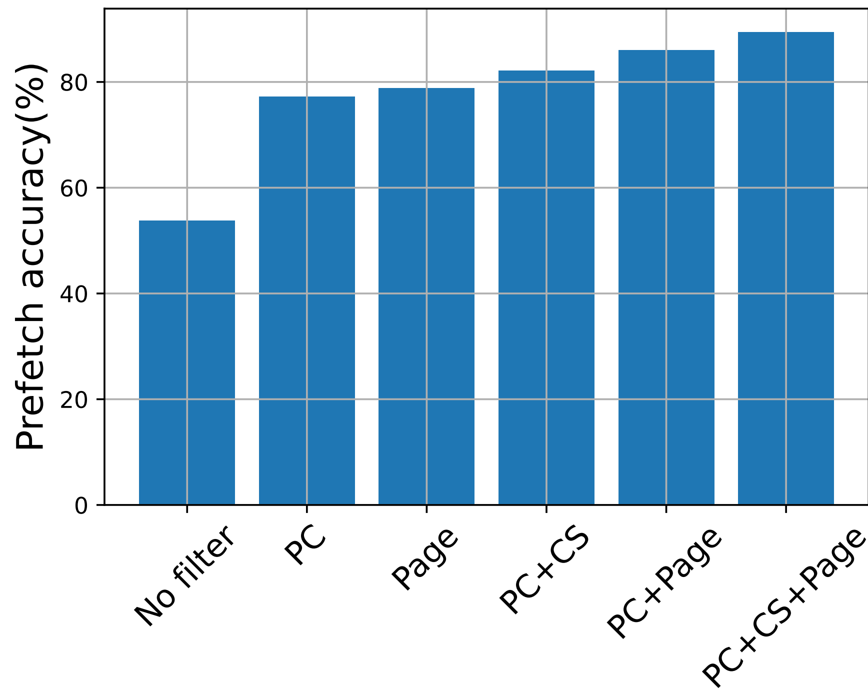


Figure 3.9: Expected prefetch accuracy if low accuracy (< 30%) features are filtered out. "No filter" is the baseline prefetcher accuracy. PC, Page and Call stacks (CS) are features that can be used independently or hierarchically to improve overall prefetch accuracy.

Page

Sometimes, the prefetching behavior of an instruction can be influenced by the object being accessed. For instance, the same proto structure can have different object sizes causing different prefetching behavior for the same PC/Call context. We tracked prefetching accuracy of individual physical pages to demonstrate pages as a feature.

We compare these features in Figure 3.9. We simulated and measured prefetching behavior across different features and predicted the accuracy improvement if we filtered the features that have prefetch accuracy <30%. Clearly, filtering based on PC provides a significant accuracy improvement opportunity

to begin with, so we focus on evaluating it in the paper.

3.4 Programmable Prefetching

This section details a concrete hardware–software system design for profile-driven programmable prefetching. As shown in Figure 3.10, our solution comprises of (1) a profiling analysis (Section 3.4.1) to find instructions that will likely experience wasteful prefetching, (2) binary instrumentation to instruct the hardware to suppress prefetching (Section 3.4.2), and microarchitectural support to act on the software-provided directives (Section 3.4.3).

3.4.1 Profiling Analysis

Our profiling phase analyzes workloads' memory access patterns to identify code locations where prefetching is particularly wasteful. In particular, in our profiler, we model a simplified memory hierarchy with a simple aggressive prefetcher to approximate the coverage and accuracy of prefetching on different code segments of a given workload.

Specifically, we analyze DynamoRio memory traces [14] consisting of all memory accesses for an execution window, including instruction fetches and load/store instructions. (This paper focuses on data prefetching; we defer instruction prefetching to future work.) Each access in the trace consists of a PC and a memory address. We then analyze the trace to get a prefetch score per instructions, and add a lightweight uarch model to improve the fidelity of the prefetch scores.

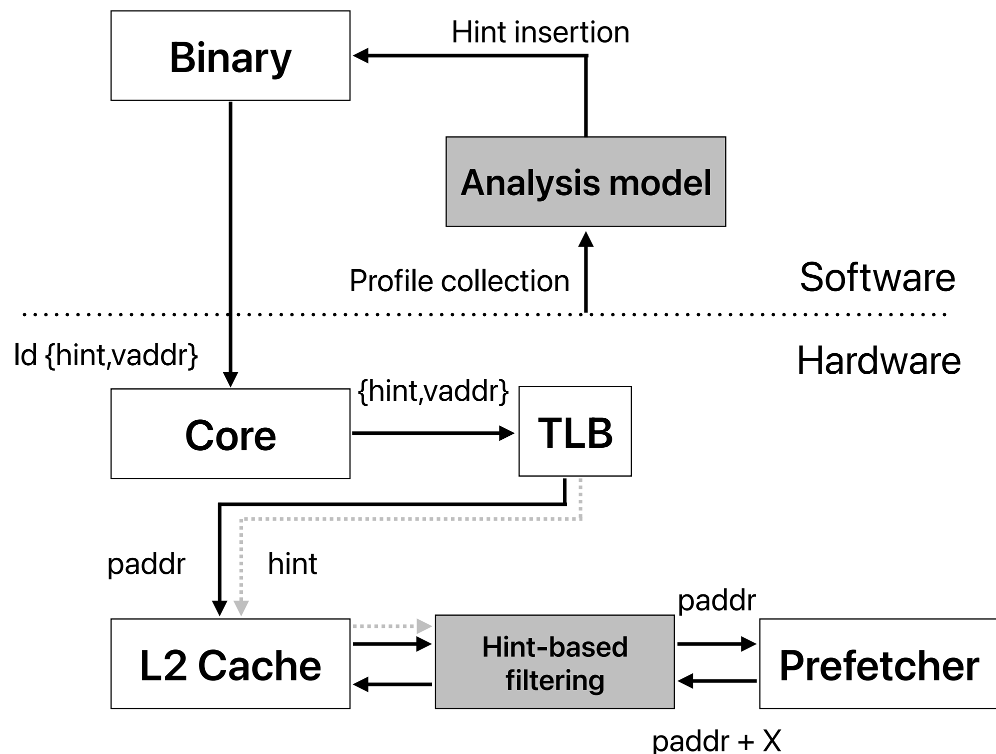


Figure 3.10: An overview of the Programmable Prefetching (ProP) system.

Prefetch scoring models

We estimate *prefetch scores* for each instruction, where an instructions is identified by a program counter (PC). Prefetch scores estimate the probability of prefetching a useful address (in the range [0, 1]) upon observing a memory access from a given instruction PC.

The prefetch coverage for an instruction is highly dependent on the underlying prefetching algorithms, but we expect that the *scoring* can be learnt more easily with an abstract prefetching model. We evaluate three different scoring techniques that model the high-level characteristics of different styles of stride prefetching, and choose a technique that is versatile and reasonably predicts the

score for multiple state-of-the-art hardware prefetchers:

- *Simple stride model:* For each memory access, we calculate the offset from the closest memory access among the most recent N requests and map it with the corresponding PC for the instruction. This results in a histogram of offsets for each PC. The prefetch score for each PC is:

  $$\text{Score}_1 = \frac{\text{Frequency of the most common offset}}{\text{Total number offset observations}}$$

  This score estimate is high for instructions that access regular strides with the same offset and low for random accesses. However, it also penalizes access patterns that oscillate between different offsets. This penalty is appropriate to model prefetchers that actually behave poorly for such an access pattern, like BOP which finds a single offset for multiple access streams. On the other hand, it is a poor approximation for prefetchers that can modulate their offsets quickly such as IP-stride prefetcher. In addition, it only considers single instruction streams for prefetching, which is very different from region-based prefetchers such as BOP or SPP.

- *Stream length model:* For each PC, we maintain a list of recent addresses observed. For each memory access, we find the smallest stride offset among the recent address list for each PC. We increment the stream length at each successive memory access with the same offset until the offset changes. Each PC has a histogram of stream lengths, which can go up to a maximum of 32. The score is defined as:

  $$\text{Score}_2 = \frac{\sum \text{frequency}_i \times \text{stream length}_i}{\sum \text{frequency}_i} \times \frac{1}{32}$$

  Instead of tracking individual offsets, this model focuses on stream length as an estimate of prefetch accuracy. In case of a short stream, due to

prefetcher warmup time and the length of the stream itself, the accuracy of the prefetcher is low, which is emulated by this model. However, this model also considers individual PC-streams and misses prefetching opportunities across streams.

- *Modeling an aggressive region prefetcher:* Whereas the previous two approaches focus on identifying per-PC streams, *region prefetchers* do not work this way: they seek offsets that fit *all* accesses within a page or code region. One example of a region prefetcher is the Best Offset Prefetcher (BOP) [62], which finds a common offset for the last *N* accesses (typically 64–256). This approach of finding a global offset generalizes well in identifying strided streams of accesses, without tracking individual delta patterns that could be biased towards a specific hardware prefetcher. However,in the case of interleaved positive and negative strides in a region, a global offset can only prefetch in one direction. We model a *dual offset* BOP-like region prefetcher that tracks a positive and negative best offset for each region, to avoid biasing the model in one direction. In this approach, we use this prefetcher model to generate prefetch candidates and associate a prefetch score with each PC based on future demand accesses using that candidate. To this end, we maintain two sets of prefetched addresses for each *directional* offset, and use it to track useful and unused prefetches (prefetched blocks that are evicted without being accessed by a demand request) for each PC. Finally, we get two prefetch accuracy metrics for a PC, and the model score is the maximum of the two:

$$\text{Score}_3 = max(\frac{\text{Useful}_1}{\text{Unused}_1}, \frac{\text{Useful}_2}{\text{Unused}_2})$$

We use the dual-offset aggressive prefetcher as the “analysis model” in Fig-

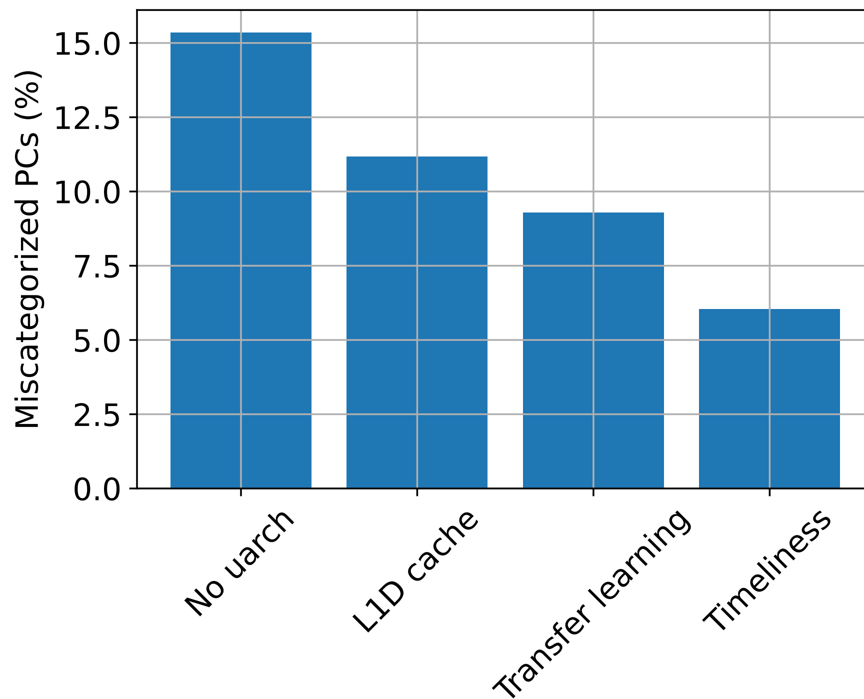


Figure 3.11: Fraction of miscategorized PC (%) based on total prefetches issued, decrease with additional level of microarchitectural modeling. The ground truth prefetch accuracy data for PCs is from gem5 simulation of `Merced`.

ure 3.10, to determine the level of prefetch filtering in hardware. In this work, we use a score threshold of 0.3, below which *all* accesses by the PC is filtered from the hardware prefetcher, thereby preventing any prefetching on it.

Modeling microarchitectural details

To ensure prefetch scores are accurate, our profiling analysis incorporates simple microarchitectural details along with the prefetch score model. For example, modeling cache filtering effects and prefetch timeliness helps better approximate the behavior of a real prefetcher without requiring the expense of a detailed simulator. Figure 3.11 shows the effect of modeling these microarchitectural parameters on miscategorized PCs, using Score_3. We define a miscatego-

rized PC as follows - A) The prefetch score from our profiling model is lower than our model threshold—30%, which would filter this PC, but the simulated hardware prefetch accuracy is above 70%, or B) prefetch score is above 30% (no prefetch filtering by ProP) but the hardware accuracy is below 30%. Each PC is weighted by the amount of prefetches issued by the hardware prefetcher, to get the final fraction(%) of miscategorized PCs. Implementing all the 3 modeling parameters reduces the miscategorization from 15.3% to 6%, in the case of `Merced`. We discuss the modeling techniques in detail below:

- *Cache modeling*: Since aggressive prefetchers are usually placed at the L2 or SLC, they see a filtered stream of memory accesses. Filtering requests at the L1 level absorbs fine-grained instruction reordering within a cache line, and filters different instructions accessing the same cache line (temporal locality). We model a simple direct-mapped L1D-sized cache to filter accesses and run the prefetch score model on accesses that miss in this cache.
- *Transfer learning for filtered instructions*: Since the prefetch model only scores unique cacheline based instructions (that filter through L1D), it can ignore instructions that are always ordered after the unique instruction accessing the same address. However modern processors have out-of-order cores which can reorder instructions. For instructions that are always filtered at L1D in the profiling stage, we maintain a frequency map of instructions that access the same cacheline first. At the end of profiling, we use the prefetch score of the top instruction in the map, as the score of the filtered instruction, since both instructions access the same cacheline and have similar memory access pattern.
- *Timeliness modeling*: In a real system, prefetches are only useful when they

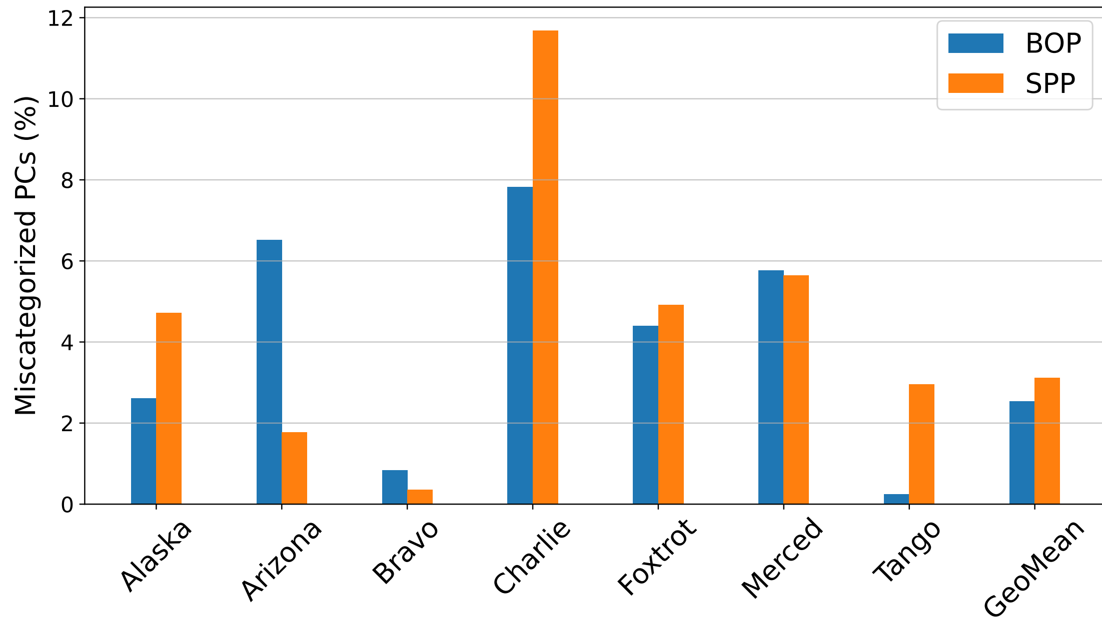


Figure 3.12: Fraction of miscategorized PC (%) by the profiling model for all datacenter workloads, for different underlying hardware prefetchers. The miscategorization error is small in all workloads, and generalizes well to both hardware prefetchers

are issued early enough to hide the latency for a subsequent demand access. For example, BOP attempts to ensure its prefetches are *timely* by using cache-fill time notifications to seek offsets that are large enough to account for the round-trip memory access latency. Large offsets come with a tradeoff of reduced coverage and accuracy, especially for short streams. However, large offsets work well for the profiling model, since it helps penalize short streams with poor prefetch accuracy scores. We model a fixed delay, of 3 memory accesses (chosen empirically) between a prefetch issue and cache fill.

Versatility of profiling analysis for different hardware prefetchers

Fig 3.12 shows the "error" of the profiling model, represented by miscategorized PCs(%), for different hardware prefetchers (BOP and SPP), across datacenter workloads. On average, the predicted scores of the model have an error of 2.5% for BOP and 3.1% for SPP, which is low for both prefetchers. Since the model is not biased towards a single prefetcher, there is oscillation in the error gap between both prefetchers on different workloads. `Arizona` has interleaved accesses of positive and negative strides, but due to BOP's limitation of a single global offset, it cannot prefetch the negative streams in this case. The profiling model can detect both streams and predicts high prefetch scores for them, causing an error of 6.5% for BOP. SPP has a low error of 1.8% because it detects offsets on a page granularity and is able to detect the two streams.

In case of `Charlie`, there are a few streams which have a stride access with alternating delta patterns of 6 and 7, which can be identified by SPP. However, BOP uses a global offset based off a list of limited offset candidates which by default, only has factors of 2,3 and 5. Even with 7 as an offset candidate, BOP is unable to find a common offset that works well for all streams in the region. So BOP is unable to prefetch accurately for this complex delta pattern. Hence in this case, our BOP-like region model, has a lower error for BOP than SPP.

Despite these inaccuracies, we find that our model is sufficiently accurate, simple and generalizable across prefetchers that it can provide hints that are useful in majority of the cases.

3.4.2 Communicating Prefetch Hints to Hardware

Our system needs a way to mark certain accesses (PC addresses) as prefetch-friendly or prefetch-unfriendly. There are two ways to do this a) an ISA extension to introduce new instructions or b) mask unused bit in the address issued by the target PC address.

In our design, we assume no changes to the ISA and use unused bit in the data address to convey this hint because prefetchers that are higher in the memory hierarchy (closer to main memory) never see the PC of the instruction that triggered a given access. Specifically, our scheme reuses the unused bits in the data address [48-63] to represent prefetch hints. For prefetch filtering, we only need 1 bit to indicate to the hardware prefetcher whether to enable or disable the hardware prefetcher for each access.

This section describes how a software mechanism can be used to instrument the binary to set the relevant address bit to convey the prefetching hint to hardware and how the hardware handles these hints.

Software instrumentation The profiler identifies the PCs for the accesses that need the hint. One approach may be to use a binary rewriting tool to instrument prefetch-unfriendly accesses to set the hint bit. We can extract the target instruction's operands and insert a platform-specific instruction to mask the upper bits and set the hint bit. The exact instrumentation depends on the addressing mode: in a register-direct addressing mode, we can directly mask the address value in the register; for other addressing modes, we first need to materialize the effective address and then apply the mask. An ISA extension alternative would have

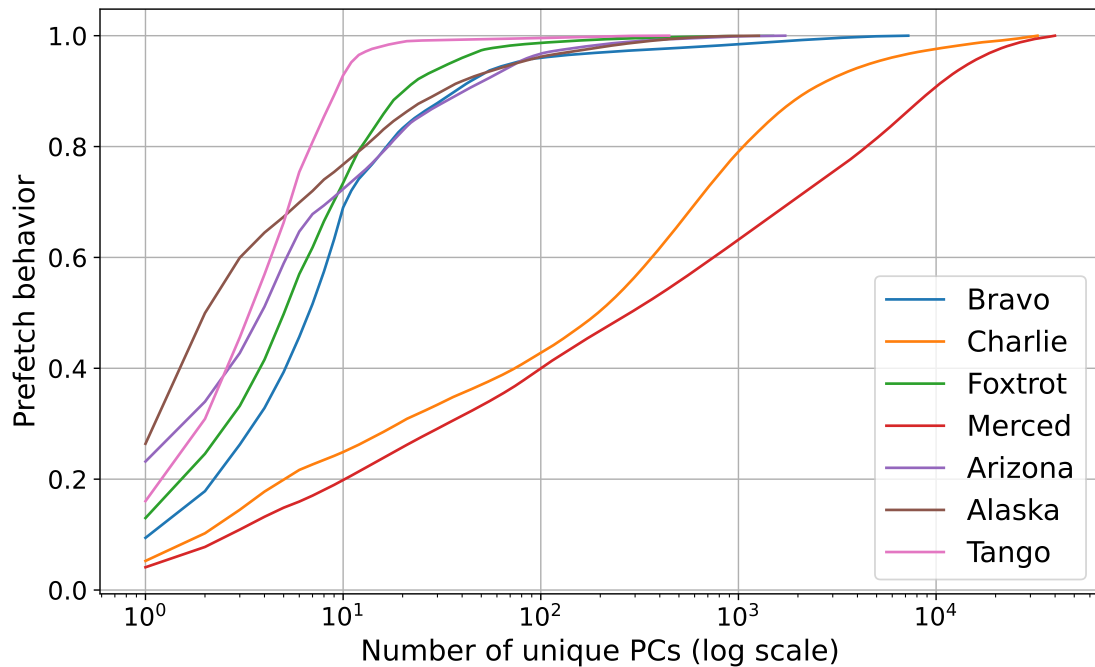


Figure 3.13: Cumulative density function (CDF) of number of unique PCs (in log scale) needed to represent prefetch behavior of datacenter workloads.

to define variants of all instructions which issue memory accesses. This may be undesirable for CISC ISAs.

The following code listing demonstrates an example of inserting a masking instruction to instrument the hint in the 48th bit location of the virtual address (of a prefetch-unfriendly access):

```
// load the addr in register
ldr r0, addr
// NEW INSTR: mask the hint inside the addr
orr r0, r0, (1 << 48)
// send the load request with hint to memory
ldr r1, [r0]
```

Figure 3.10 shows the resulting masked representation of the virtual address as {hint,vaddr}.

To quantify how many instructions are affected, Figure 3.13 shows the number of unique PCs that represent all prefetch behavior (total prefetches issued by all PCs) of the workload. We instrument profile hints for a fraction of these PCs, which are below a prefetch accuracy threshold. For most workloads except `Merced` and `Charlie`, ~10 PCs can represent 80% of the prefetching behavior. For the larger workloads, ~5K PCs represents 80% of the behavior. This is a much smaller fraction of the entire unique instruction footprint of the workload, which for `Merced` for instance, is >1M instructions.

While we do not consider it in this work, it would be possible to eliminate the overhead of these instrumentation instructions by extending the ISA. This extension would add hint bits to memory access instructions, much like NTA hints. Like NTA, the hints would get plumbed into the memory subsystem to provide prefetching directives.

Address translation Both Intel's Linear Address Masking (LAM) [39] and Arm's Top Byte Ignore (TBI) [1], currently reserve a few bits that are unused in the address space. This allows application software to store metadata in the higher order bits. This hint is embedded in the virtual memory address and is ignored during the virtual to physical address translation. In our methodology, as shown in Figure 3.10, we propose extracting the hint bit from the virtual address, and retaining it with the physical address output of the TLB, in the request packet. The hint is retained in the request packet until it reaches the prefetch filtering block.

Parameter	Value
CPU core	1-4 cores, OoO, 3GHz, 320 entry ROB, 90 LSQ
Private L1	Split I/D, 64KB, 4-way, 2 cycle, PLRU
Private L2	1MB, 8-way, 9 cycle, PLRU, inclusive
Shared L3	1MB/core, 16-way, 29-33 cycle, PLRU, partially inclusive
Main Memory	Load-to-use latency: 95ns, 4 GB/s (single-core), 16GB/s (4-core), 32GB/s (8-core)

Table 3.1: Simulation configuration

3.4.3 Prefetch Filtering in Hardware

Figure 3.10 shows the "hint-based filtering" block, which is placed between the L2 cache and the underlying prefetcher at that level. The block filters the addresses based on the hint bit associated with it. Hence, the prefetcher only observes a filtered stream of prefetch-friendly accesses. In addition to reducing prefetch traffic, such *proactive* filtering at the input of the prefetcher also improves the offset detection algorithm due to an improved stream visibility.

As evident from Figure 3.10, the hardware changes are minimal and only require minimal logic to interpret the hint bit.

3.5 Methodology

3.5.1 Performance Model

We use gem5 simulator [12], an event-driven simulation model. Table 3.1 summarizes the CPU and memory parameters of the simulated model for both single-core and multi-core simulations. We use memory bandwidth of 4GB/s for our single-core experiments based on recent trends for per-core memory

Name	Thread count, average size	Simulation window
`Alaska`	5 threads, 1B	[400M, 800M]
`Arizona`	5 threads, 7B	[400M, 800M]
`Bravo`	4 threads, 1.1B	[400M, 800M]
`Charlie`	5 threads, 2.2B	[400M, 800M]
`Foxtrot`	5 threads, 3.4B	[400M, 800M]
`Merced`	5 threads, 850M	[400M, 800M]
`Tango`	3 threads, 1.3B	[400M, 800M]
SPEC 2017	10 benchmarks, 300B	4668 simpoints, 10M each

Table 3.2: Workload simulation details

bandwidth in server systems [104, 42]. We scale out system bandwidth to 16GB/s for 4-core, and 32GB/s for 8-core evaluation.

3.5.2 Workloads

We simulate 7 datacenter workloads and 10 SPEC2017 INT benchmarks for our evaluation using their respective DynamoRIO traces. The workloads are listed in Table 3.2. The datacenter workloads have been provided codenames to preserve anonymity.

The datacenter workloads have profile-guided and cross-module thinLTO optimizations that enable deep callsite inlining behavior. The SPEC workloads have the standard compiler optimizations as recommended on the benchmark website.

Single-core simulation: Each datacenter workload has multiple threads, and we simulate the top 5 threads based on instruction count, such that each thread has at least 800 million instructions. For each thread, we fast-forward the first 300M instructions, use the next 100 million instructions to warmup the

cache, and we perform a detailed simulation of 400M instructions in the range (400M,800M).

For SPEC workloads, we simulate 4668 10M-interval Simpoints [79] that represent the entire trace of the benchmark, effectively simulating ~4.6B instructions for each SPEC workload. Before each simpoint execution, we warmup the cache by running 30M instructions prior to the start of the simulation.

Multi-core simulation: For multi-core, we simulate 4-core and 8-core configurations. We generate 100 random mixes of datacenter workload threads. We fast-forward the first 300M instructions, warmup for 100M instructions, and perform a detailed simulation for the next 100M instructions. For each simulation, once a workload finishes execution on a core, we stop accounting it's stats but keep it running until all other workloads have finished execution.

We report performance as a weighted speedup over no prefetching baseline. First, for each prefetcher configuration of a workload in the multi-core mix, we calculate IPC of the workload when run together with all other workloads - $IPC_{together}$. In addition, we run the same workload without prefetching, alone, on a 4-core system, to calculate IPC_{alone}. We calculate the weighted IPC of a workload mix as the sum of individually *normalized* IPC of each workload *i*: $\sum_i$ $(IPC_{together}/IPC_{alone})$. Next, we similarly calculate weighted IPC for the workload mix without prefetcher as a baseline. The reported weighted speedup of the workload is obtained by normalizing the weighted IPC with prefetcher enabled, to the weighted IPC without prefetching i.e., the baseline.

For the 8-core configuration, we use the same methodology as 4-core, but reduce the warmup to 50M instructions and perform detailed simulation for

the next 50M instructions. This reduction is done to finish the experiments in a reasonable time frame (~ 2 days).

Short-instruction window simulation: As shown in Figure 3.3a, context switches are frequent in datacenters and can occur every 250K-500K instructions. To evaluate the impact of frequent context switches, we simulate a range of instruction window lengths such as 250K, 500K, 1M, 10M and 50M, and warmup by 50M instructions before to decouple prefetcher performance from cache warmup. For a single window length such as 250K instructions, we simulate all such windows in the range [400M, 450M] (200 windows in this case), for all datacenter workloads. For each workload, we average IPC across the N windows for all threads of the workload, to get the final IPC performance.

3.5.3 Profiling methodology

We use the "aggressive region prefetcher" model for profiling traces to generate hints. By default, we choose a threshold score of 30% accuracy to determine which load instructions should be filtered. For datacenter traces, we profile the first 300M instructions for all the threads and pool the data across threads of a given workload, to generate one set of hints per workload. For, SPEC 2017 INT benchmarks we profile 1B instructions in the range of (1B,2B) for each workload. Our profiling methodology ensures that the training region for each workload trace (part of the trace used to generate the profile) does not overlap with the test region (part of the trace that is simulated for performance measurements), or has very low probability of overlap in the case of SPEC. Note that we do not evaluate cross-trace generalization as it is common for datacenters to have

continuous profiling for key web services [92, 82, 108].

3.5.4 Baselines

We use 2 prefetchers and 2 hardware throttling techniques for data prefetching at L2 as our baselines. For baseline prefetchers, we choose two state-of-the-art prefetcher models: Best Offset Prefetcher (BOP) [62] and Signature Path Prefetcher (SPP) [48]. BOP was the winner of 2nd Data Prefetching Championship. BOP uses minimal hardware resources and issues prefetches with a global offset, while also taking into account prefetch timeliness. SPP adds support to track individual delta patterns and has been shown to perform better than BOP on SPEC 2006 and 2017 benchmark [48, 11]. Each prefetcher's parameters are tuned to maximize their performance in the baseline system. For SPP, this corresponds to a 25% confidence threshold, and for PPF-SPP, this corresponds to a 0% threshold.

We use Feedback Direceted Prefetching (FDP) [107] as a coarse-grained throttling mechanism, and Perceptron-based Prefetch Filtering (PPF) for fine-grained filtering. FDP was originally designed to work with a stream prefetcher [44] with variable prefetching degree. Since BOP already operates at an aggressive offset it only uses a degree of 1. We model FDP to reduce the degree of BOP from 1 to 0, effectively stopping prefetching in regions with poor prefetcher accuracy. We use the default configurations for PPF, which was proposed to filter prefetches issued by an aggressive version of SPP (with no confidence-based throttling) based on a perceptron model.

We demonstrate ProP's ability to filter prefetches for *both* prefetchers: BOP

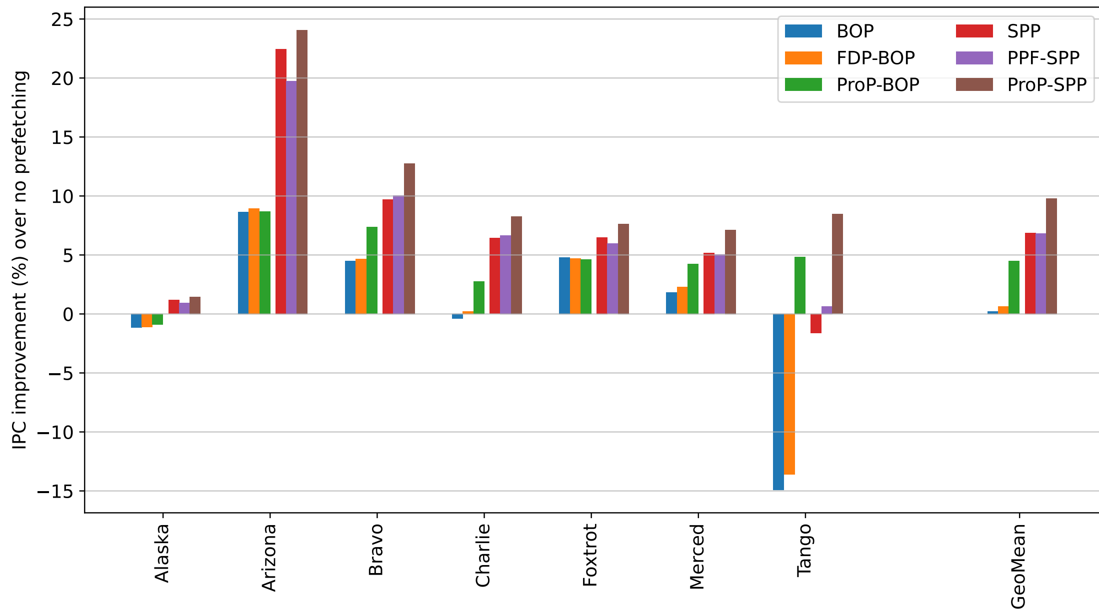


Figure 3.14: Single core IPC improvement over no prefetching for all datacenter workloads at 4GB/s bandwidth

and SPP. Since ProP is proactive (not reactive) in filtering prefetches, we use profile-guided hints embedded in the data address to filter accesses. If the data access is a bad candidate for prefetching, we filter the accesses from both training and iprediction. In addition, for ProP-SPP, we reduce the confidence thresholding in SPP (from the default of 25% to 5%) to maximize coverage and use profiling hints to improve accuracy. We use the 5% threshold to avoid prefetching on randomly observed deltas for a PC.

3.6 Evaluation

3.6.1 Single core results

Figure 3.14 shows that ProP benefits both BOP and SPP. In particular, ProP-BOP achieves an IPC improvement of 4.5% over no prefetching, while the underlying prefetcher (BOP) achieves an IPC improvement of just 0.2%. Coarse-grained throttling with FDP is not very effective as it achieves an IPC improvement of just 0.7%. ProP-SPP achieves an IPC improvement of 9.8%, that is 3.9% higher than SPP, and 4% higher than PPF.

Both, ProP-BOP and ProP-SPP either match or outperform their respective baselines for all the workloads. Workloads with interleaved streams of random accesses and stride patterns, such as `Tango`, benefit the most from ProP. ProP increases prefetch accuracy by filtering instructions with poor prefetchability, which reduces bandwidth contention and cache pollution. In particular, for `Tango`, ProP-BOP increases the prefetcher accuracy of BOP from 52% to 79%, which reduces the memory traffic overhead of BOP from 62% to 11%. Similarly, ProP-SPP has an L3 traffic overhead of just 15%, while PPF-SPP has a 30% traffic overhead. At the same time, ProP-SPP has a coverage of 52.7%, which is 4.3% higher than PPF-SPP. Hardware prefetch filtering techniques such as FDP-BOP and PPF-SPP have limited precision parameters and thresholds that are tuned for SPEC workloads, but do not generalize for unseen high instruction footprint workloads in the datacenter.

Accuracy-coverage tradeoff: Figure 3.15 shows the accuracy-coverage tradeoff for all prefetcher configurations averaged across all datacenter workloads.

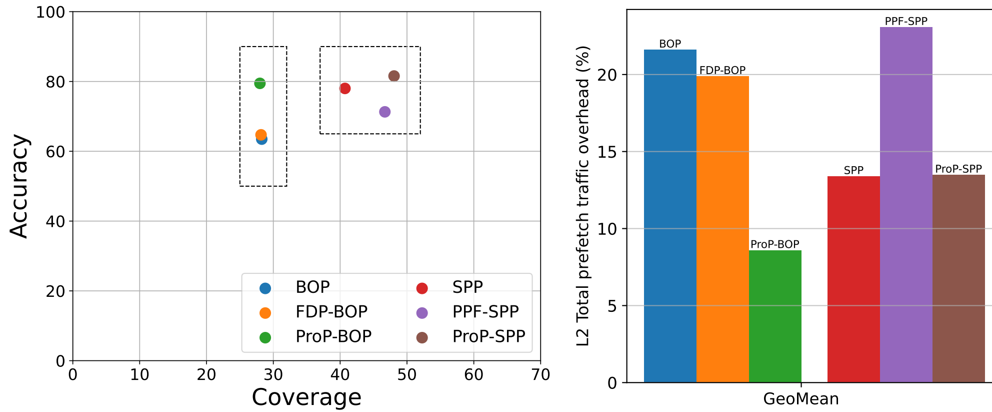
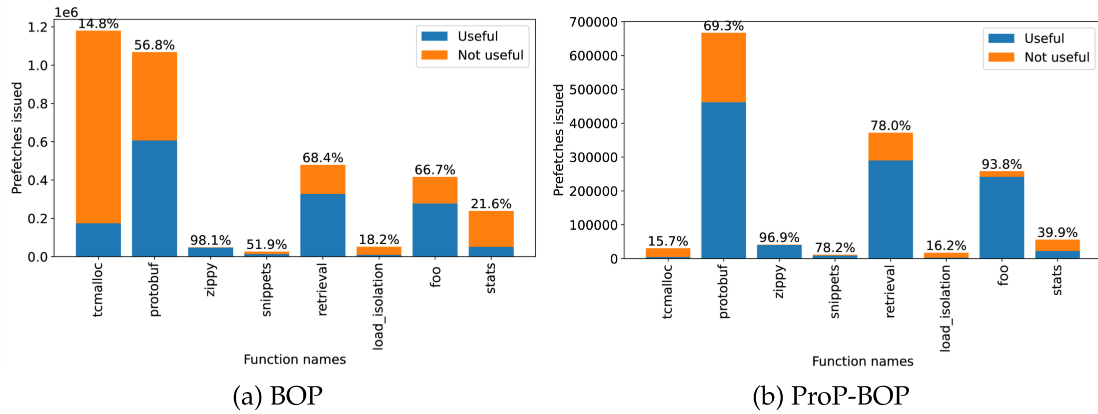


Figure 3.15: Accuracy-Coverage tradeoff of various prefetcher configurations, averaged across all datacenter workloads. ProP-BOP and ProP-SPP allows better coverage and accuracy tradeoff, and low traffic overhead.

ProP-BOP improves the accuracy of BOP to 79.5%, which is 16% higher than BOP, while maintaining the same coverage. This accuracy improvement is also reflected in the L2 traffic overhead reduction. ProP-BOP can reduce traffic overhead by 11% over FDP-BOP.

ProP-SPP improves the accuracy over SPP by 3.5% *and* coverage by 7.3%. It may seem surprising that ProP-SPP improves coverage since it only filters prefetches, but as we note in Section 3.5.4, both ProP-SPP and PPF-SPP use a confidence threshold that maximizes performance. The lower thresholds result in a coverage improvement (discussed in Section 3.5.4). In particular, because ProP leverages profiling to filter bad prefetches, it does best with a lower confidence threshold that allows it to maximize the opportunity for useful prefetching. The profile guided filtering compensates for the accuracy. By contrast, PPF-SPP's increase in coverage comes at a cost of lower accuracy. This improved tradeoff is reflected in L2 traffic overhead, where ProP-SPP reduces traffic by 9.5% over PPF-SPP. Overall, ProP allows a better accuracy-coverage tradeoff for the underlying prefetcher, while reducing memory bandwidth contention.



(a) BOP (b) ProP-BOP

Figure 3.16: Library-level prefetch accuracies with baseline BOP and ProP-BOP.

Library-level performance improvement: Figure 3.16 shows the effect of ProP on prefetching behavior for different libraries for the `Merced` workload. ProP improves the prefetch accuracy of all libraries. Even though ProP can effectively filter at instruction-granularity, certain libraries such as `TCMalloc` and `load_isolation` are not prefetch friendly, and get throttled to preserve high accuracy prefetching. Other libraries such as `foo` observe a significant accuracy improvement due to finer-grained filtering of instructions.

SPEC 2017 INT benchmarks: Figure 3.17 shows the IPC improvement results for all benchmarks in the SPEC 2017 INT suite. In addition to geomean of all benchmarks, it also shows the geomean for the memory intensive workloads. On average, for memory intensive benchmarks, ProP-BOP achieves an IPC improvement of 10.8%, which is 2.7% higher than BOP and 0.7% higher than FDP-BOP. ProP-SPP achieves an IPC improvement of 12.7%, which is 3.3% higher than SPP and 1.2% higher than PPF-SPP. ProP outperforms the underlying prefetchers in most cases and we discuss a few interesting ones here. `520.omnetpp` observes a substantial performance increment of 6% by using ProP-BOP over BOP, which regresses performance. BOP has a 75% L2 traffic overhead due

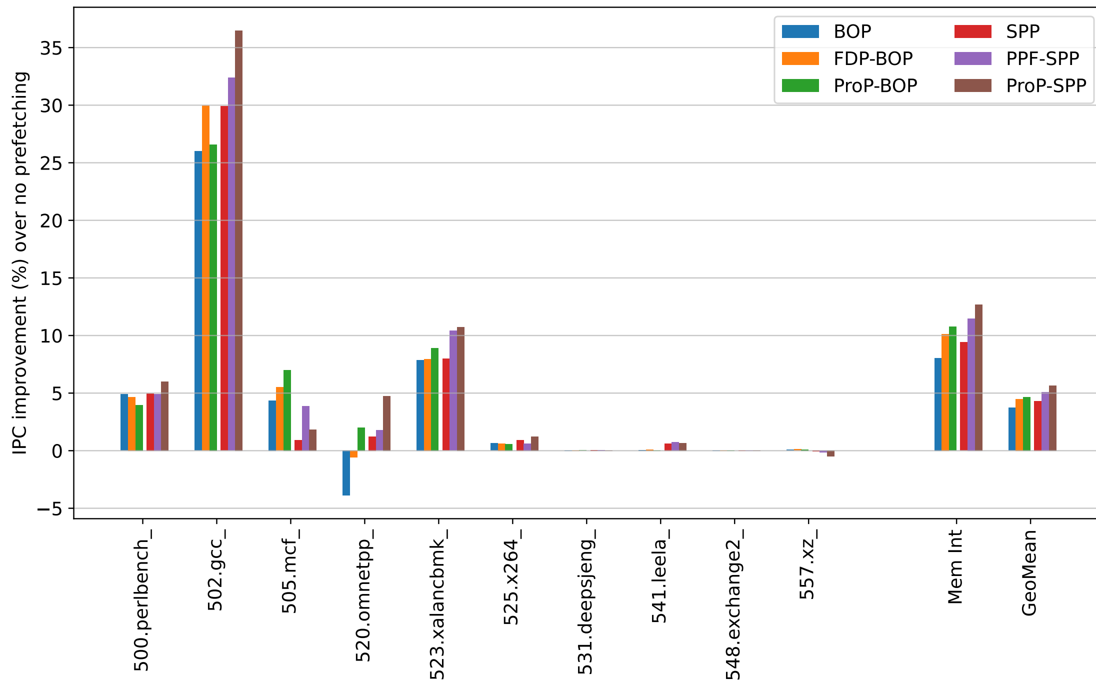


Figure 3.17: Single core IPC improvement over no prefetching for SPEC workloads at 4GB/s bandwidth

to poor accuracy, and ProP-BOP reduces it to 27%, while improving the coverage slightly. FDP-BOP also throttles the prefetcher but due to uniform throttling for a region, it reduces the coverage from 11% in ProP-BOP to 4.6%. `505.mcf` is an interesting case since BOP outperforms SPP, even though SPP has a higher coverage. We find that BOP issues more *timely* prefetches compared to SPP, resulting in a reduction in effective memory read latency. In this case, ProP-BOP outperforms all prefetcher configurations including ProP-SPP.

3.6.2 Multi-core results

4-core: Figure 3.18 shows the IPC speedup of 100 groups of datacenter workloads on a 4-core system. The groups are sorted in order of increasing performance on ProP-SPP. ProP-SPP outperforms the other two configurations -

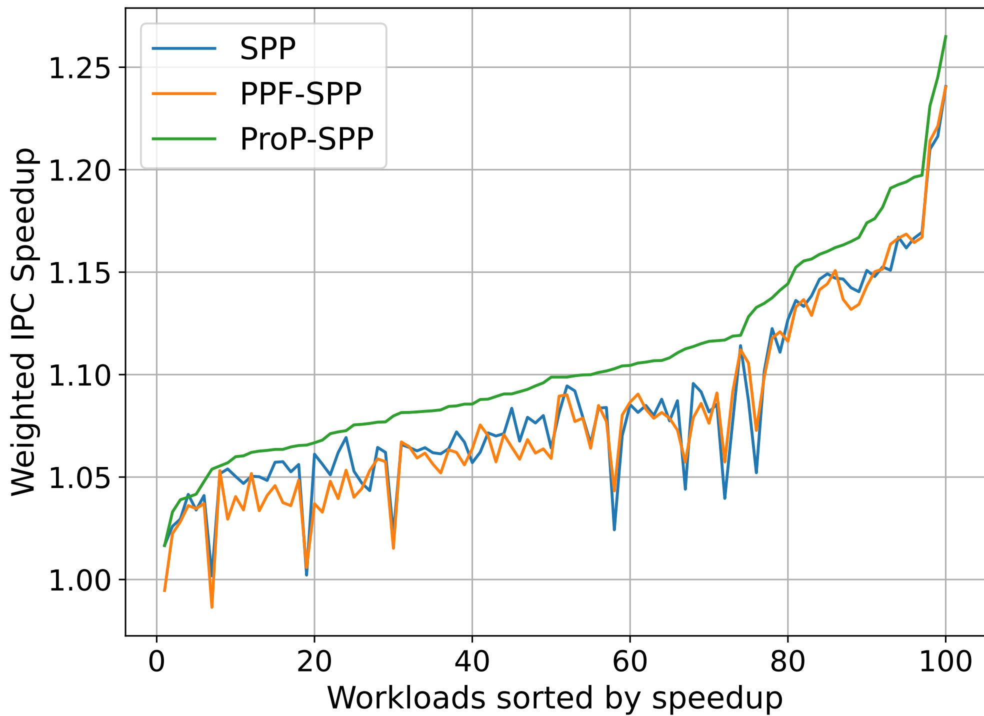


Figure 3.18: IPC speedup for 100 datacenter workload mixes in a 4-core setting at 16GB/s system bandwidth.

SPP and PPF-SPP in each of the 100 mixes. On average, ProP-SPP achieves a speedup of 10.5%, which is 2.6% higher than PPF and 2.2% higher than SPP.

8-core: Figure 3.19 shows the IPC speedup of 100 groups of datacenter workloads on a 8-core system. ProP-SPP outperforms the other two configurations - SPP and PPF-SPP in each of the 100 mixes. On average, ProP-SPP achieves a speedup of 9.3%, which is 2.6% higher than PPF and 2.9% higher than SPP.

3.6.3 Context-switch sensitivity

Datacenter workloads often context switch within short instruction window intervals, as discussed in the Section 3.2. Figure 3.20 evaluates average perfor-

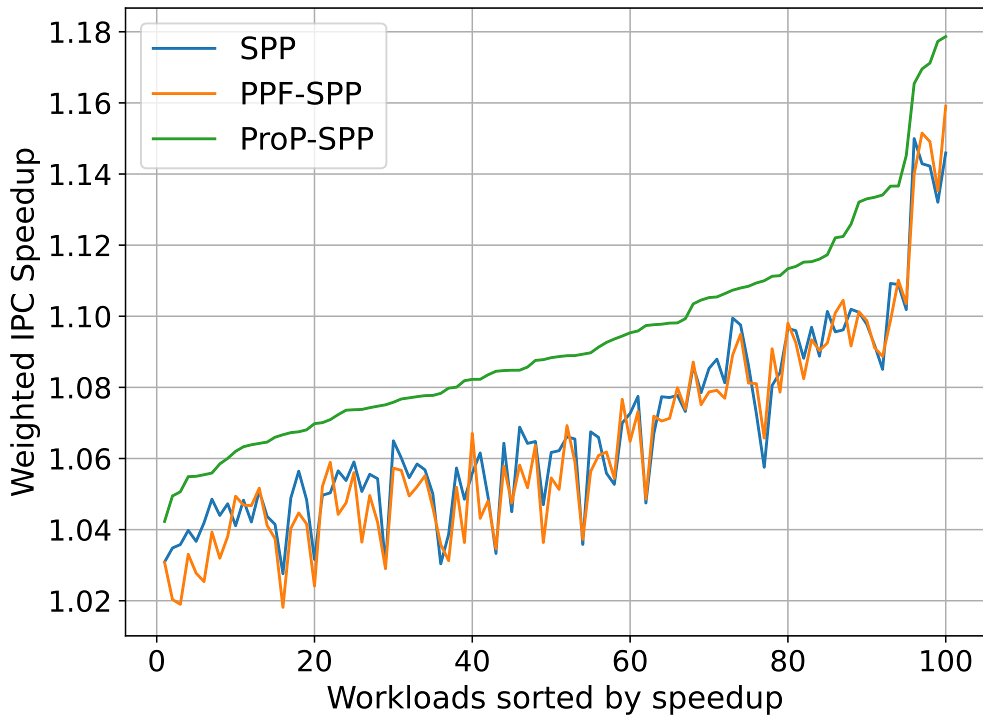


Figure 3.19: IPC speedup for 100 datacenter workload mixes in a 8-core setting at 32GB/s system bandwidth.

mance on datacenter workloads at 4GB/s, of SPP, PPF-SPP, ProP-SPP and SPP-aggressive prefetchers, across different window lengths. SPP-aggressive has no confidence threshold, unlike the default SPP with 25% threshold, and serves as a configuration with no prefetch throttling/filtering. For a small instruction window such as 500K, ProP-SPP improves IPC by 8.1% over the baseline, which is 2.4% higher than SPP and 3.4% higher than PPF-SPP.

Figure 3.21a shows the traffic overhead of these prefetcher configurations for different window lengths. For all prefetchers, L2 traffic overhead increases with instruction interval, which can be attributed to warmup time of SPP. At 250K, even SPP-aggressive has a low traffic overhead due to limited prefetching opportunities identified by the prefetcher. PPF-SPP takes a while to reduce L2 traffic overhead significantly compared to SPP-aggressive (no filtering). Fig-

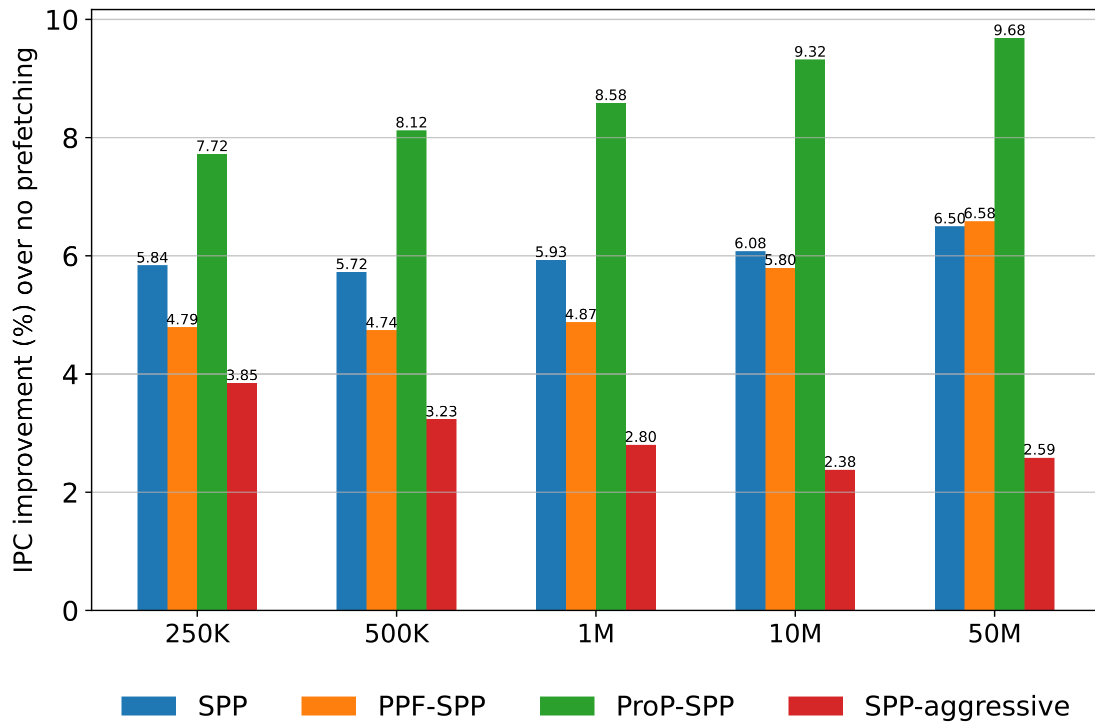
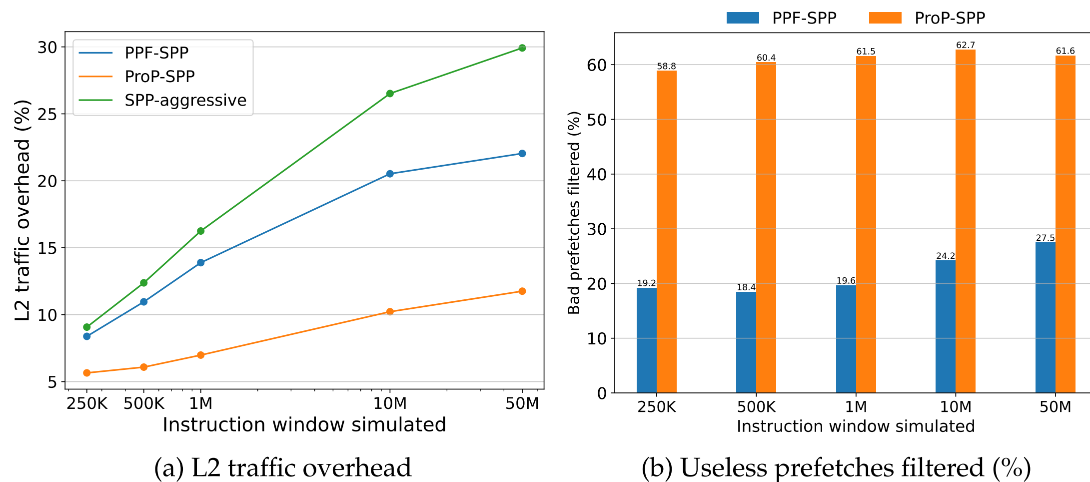


Figure 3.20: IPC speedup averaged for all datacenter workloads, simulated at different instruction window lengths.



(a) L2 traffic overhead

(b) Useless prefetches filtered (%)

Figure 3.21: Prefetch traffic and prefetch filtering efectiveness of different prefetchers at different instruction window sizes.

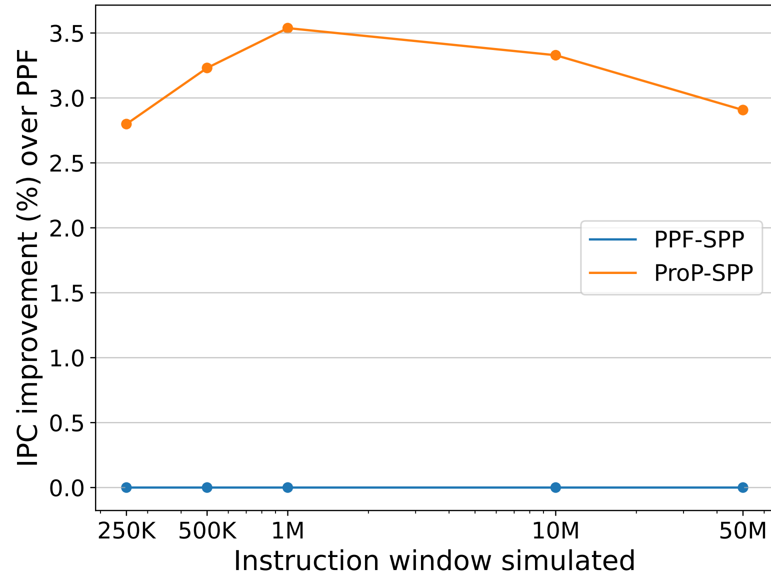


Figure 3.22: IPC improvement (%) of ProP-SPP over PPF-SPP across different instruction windows.

ure 3.21b shows PPF's filtering increases after 1M instructions. However, since traffic overhead of underlying prefetcher at small window lengths is itself low, the penalty of poor filtering on IPC is low as well. Due to this coupling effect, the IPC performance of PPF-SPP is not greatly affected at the smallest instruction window. Figure 3.22 shows the IPC performance gap between ProP-SPP (which needs no online learning) and PPF-SPP increases with larger window sizes, and is the highest at 1M instructions (3.7%), and decreases later on as PPF is able to learn. Since ProP does not need online training time, it's performance increases on every instruction window with increase in underlying prefetcher activity. In addition, it will continue to perform well with prefetchers that have a smaller warmup time.

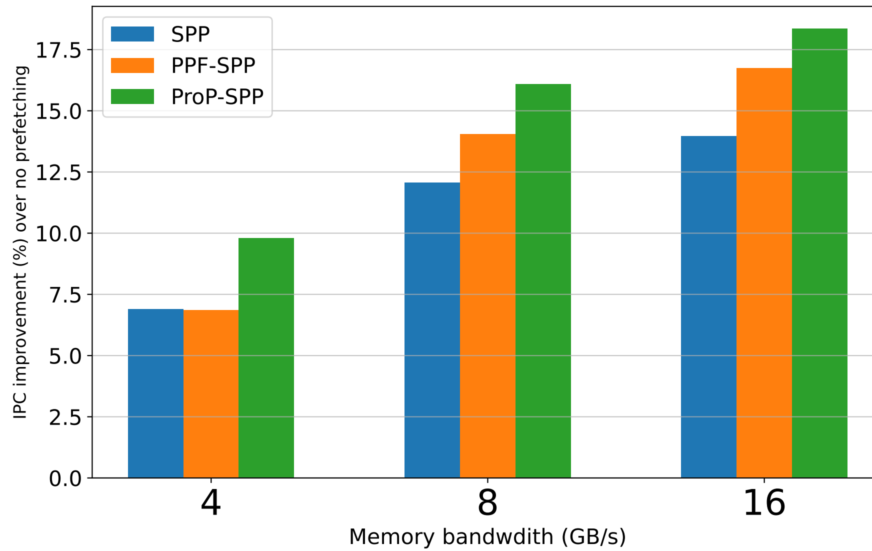


Figure 3.23: IPC improvement over no prefetching on all datacenter workloads, for SPP, PPF and ProP at different system bandwidths.

3.6.4 Bandwidth sensitivity

In addition to results on the default system bandwidth of 4GB/s, we also measured performance on higher bandwidth availability for a single-core. Figure 3.23 shows IPC speedup over no prefetching for SPP, PPF-SPP and ProP-SPP averaged across all datacenter workloads, measured at different system bandwidth configurations. At high bandwidth of 16GB/s, ProP-SPP improves performance by 4.4% over SPP and by 1.6% over PPF-SPP. This increase in performance at higher bandwidths is due to the higher coverage of ProP-SPP by lowering the confidence thresholds of SPP, enabled due to accurate profile-guided hints.

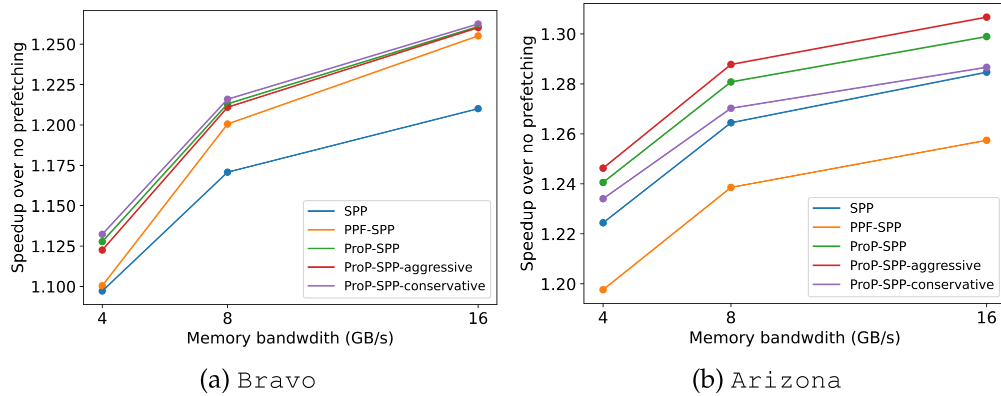


Figure 3.24: Workload-specific speedup for different prfetchers and profiling thresholds of ProP (default, aggressive, conservative), across bandwidth configurations.

3.6.5 Workload-specific profile-model tuning

Figure 3.24 shows performance of different profiling model configurations across system bandwidths. We vary the prefetch score threshold of the model, below which the hardware prefetcher filters requests. The thresholds for this experiment are 20% (aggressive), 30% (default) and 50% (conservative). Lower threshold (ProP-SPP-aggressive) corresponds to lesser filtering and can often maximizes coverage at the cost of accuracy. On average, ProP-SPP-aggressive is the best performing configuration at high system bandwidth (16GB/s) and achieves a 19% IPC improvement, which is 0.6% higher than ProP-SPP. For `Bravo`, higher degree of filtering also improves prefetch coverage, which makes ProP-SPP-aggressive perform well in all bandwidth configurations. Similarly, `Arizona` benefits from ProP-SPP-conservative irrespective of system bandwidth. Since ProP allows software control of prefetching, we can easily tune the model to account for workload specific behavior to maximize performance.

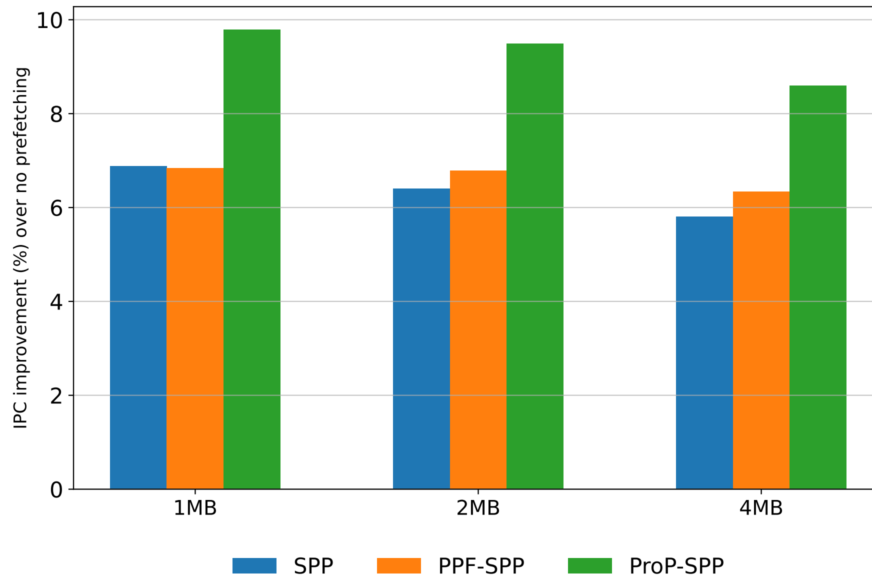


Figure 3.25: IPC improvement over no prefetching on all datacenter workloads, for SPP, PPF and ProP at different SLC sizes/core.

3.6.6 Different LLC sizes

Figure 3.25 shows the performance of prefetchers at different SLC sizes per core. ProP-SPP outperforms SPP and PPF-SPP in all SLC configurations. At higher SLC sizes, the no prefetching baseline performance improves and diminishes the headroom for prefetcher improvements, which results in decrease in IPC improvements for all prefetching configurations. PPF-SPP performs slightly better than SPP at higher L3 sizes, because the importance of prefetcher coverage increases relative to accuracy. Even at 4MB/core, ProP-SPP outperforms other configuration and improves IPC performance by 8.6%, which is 2.3% higher than the next best configuration: PPF-SPP. Overall, the effectiveness of ProP-SPP increases with smaller LLC/core configuration, due to higher prefetcher coverage and accuracy.

3.7 Related Work

Existing prefetching solutions can be divided into 3 categories: hardware prefetching, software prefetching and hardware-software co-design solutions.

Hardware prefetching: Section 1.2.2 discusses coverage-optimized and bandwidth-efficient prefetchers in detail. We compare our work with throttling-based bandwidth-efficient prefetching solutions that improve underlying prefetcher accuracy by filtering prefetches. On the other hand, our method should be compatible with composite-based bandwidth-efficient prefetchers, where we could perform patter classification in the compiler and enable sub-prefetchers using the classification hints.

Adaptive stream detection (ASD) [37] measures stream length histogram (SLH) for a program region (every few instruction epochs), and uses it to filter prefetching. Since SLH is calculated for the aggregated region, individual streams get filtered uniformly irrespective of their differences. In addition, there is a warmup time associated with learning SLH.

FDP [107] uses dynamic metrics such as prefetcher accuracy and cache pollution to control aggressiveness of the prefetcher. Similar to ASD, it uniformly throttles prefetching based on the average accuracy of the prefetcher, which results in coverage loss.

Perceptron filter (PPF) [11] is a hardware-based fine-grained prefetch filtering mechanism for Signature Path Prefetcher (SPP). They train using a set of 9 features based on page addresses, program counter and other SPP metrics such as confidence, using ~40KB on hardware, to filter SPP candidates. While hard-

ware filtering techniques have been shown to improve accuracy of underlying prefetchers, they suffer from having a warmup time, large on-chip storage and are fine-tuned to work well for a fixed set of benchmarks.

Software prefetching: Early software prefetching [17] used static induction variable analysis to prefetch stride patterns inside loops. Mowry et al. [66] extended it by identifying highly reused addresses to limit prefetching, and using loop splitting to prefetch ahead without branch conditions. Ainsworth and Jones [4] proposed automatic compiler-injected prefetching for more complex irregular accesses. APT-GET [43] improved timeliness of these prefetches by profiling Intel PMU counters.

Software prefetching methods have promise in niche memory access patterns such as irregular accesses. These methods could work in tandem with ProP, which is focused on improving hardware prefetcher accuracy for *all* prefetches, that sotfware prefetching might not cover.

Hardware-software co-design: Guided-region prefetching (GRP) [113] uses compiler static-analysis to detect loop bounds of load instructions. This is used to guide the underlying region prefetcher [56] to limit the size of prefetching region and reduce traffic overhead. GRP, unlike ProP, relies on static analysis which can only find a small subset of strided instructions in large scale workloads with limited function inlining and complex control flow.

Efficient content-data prefetching (ECDP) [26] uses profile-guided hints to score all possible prefetch offsets in a physical page, for each pointer-based instruction. ECDP focuses on a domain-specific prefetcher [23] for linked data structures (LDS), and requires a finer-grained one-hot encoded 16-bit hint

for each instruction. ProP is lightweight and versatile across state-of-the-art prefetchers for regular and complex stride prefetching.

3.8 Conclusion

Hardware prefetchers are essential in backend-bound datacenter workloads but often regress performance due to high bandwidth consumption, in an environment with limited bandwidth availability. Hardware-software co-design of prefetchers allows for a better division of responsibilities, compared to hardware-only solutions. This co-design enables ProP to find a universally better tradeoff point by improving both prefetcher metrics- accuracy and coverage.

CHAPTER 4

CONCLUSION AND FUTURE DIRECTIONS

Chapter 2 discusses improvements for auto-vectorization in compilers for emerging vector ISAs. It also proposes ScaleIR to improve compiler code generation and create avenues for more vectorization opportunities. Chapter 3 uses compiler and profile-guided hints to improve hardware prefetcher performance in bandwidth-constrained environments, often observed in datacenter workloads. It designs and evaluates ProP, which can improve both prefetch accuracy and coverage of state-of-the-art hardware prefetchers, and outperform hardware-based prefetch filtering techniques.

In the next few sections, we discuss potential areas of future work that are inspired from Chapter 2 and Chapter 3 results.

4.1 Length-agnostic speculative vectorization

Compiler vectorization techniques often embed fixed vector-length values to determine vectorizability of loops. There are several such factors affecting vectorization such as static cross-iteration dependencies and cost-profitability analysis. However, length-agnostic vector extensions abstract-out the hardware vector length. Since vector length is unknown at compile-time, the compiler is forced to deem loops which *depend on vector length guarantees* as not vectorizable. This is the Faustian bargain that VLA makes: you gain portability, but you sacrifice specificity!

We discuss an example of a loop pattern with cross-iteration dependencies:

```
for (int i = 8; i < N; i++) {
```

```
    b[i] = b[i - 8] + a[i]; // vectorizable for VLEN <= 8
}
```

We observe that the code can be vectorized when hardware vector length is <= 8, but auto-vectorization on wider machines would lead to incorrect results. A traditional compiler for length-agnostic ISAs such as RVV, would *always* choose to *not* vectorize it, making it worse than fixed-length ISAs. We envision AdaVec, which could generate optimal code for both ranges:

- Perform compiler analysis to determine a vector-length constraint. This constraint is resolved dynamically and leads to two execution paths.
- Optimize both the paths independently. If the constraint is true, vectorize the loop with the compile-time unknown hardware vector length. Otherwise, depending on the ISA support and the constraint, configure vector lanes to conform to the constraint or execute scalar code. RVV provides an opportunity to use `vsetvl` instructions to dynamically configure the hardware vector length. In other length-agnostic ISAs such as ARM SVE, predication can be useful to switch off some vector lanes.

We illustrate vectorization of the loop pattern above, using `AdaVec` in length-agnostic RVV setting:

```
// dynamic hardware vector length
vlen = get_hw_vlen();
if (vlen<=8){ // determine constraint
    for (int i = 8; i < N; i+=vlen) {
        b[i:i + vlen] = b[i - 8:i - 8 + vlen] + a[i:i + vlen];
    }
```

```
}
else{
    //set vector length to 8
    set_vl(8);
    for (int i = 8; i < N; i+=8) {
        b[i:i + 8] = b[i - 8:i] + a[i:i + 8];
    }
}
```

AdaVec remains length agnostic i.e., portable across all vector hardware, but also gets the benefit of length-specific vectorization. Since the speculation happens outside the loop, the overhead of AdaVec is negligible compared to the vectorization benefits.

A significant contribution of `AdaVec` would be decoupling vectorization analysis from the hardware vector length. This would entail finding vector-length constraint for each analysis and composing them together. A good baseline for finding constaints could be iterating over various vector lengths and keeping the lowest-cost non-identical variants. A more sophisticated version could some analysis feedback from the vectorizer. It would also be interesting to explore dynamically configured hardware vector length to conform to a constraint. This technique combines the benefits of user-configured fixed-width vectors with scalability of length-agnostic ISAs.

Applications such as fixed-length tiling in matrix multiplications or window-buffering in 3x3 convolution filters, would benefit from AdaVec's adaptive approach.

4.2 Programming model for scalable vectors

There are a range of existing vector programming models with varying degrees of programmer control and intrinsic abstraction. Programming models such as OpenMP [93] and Cilk [95] are developed for multi-core programming instead of SIMD vectorization and provide implicit vectorization opportunities using `pragma`. SIMD-specific C++ extensions such as Boost [29], provide templated vector data structure for ISA portability and extend libraries to support special functions such as "shifting" operators common in image processing workloads, but programmer effort is required to explictly vectorize the code. Sierra [55] allows programmer to program at a higher level by automatically vectorizing control flow statements using mask generation and providing memory layout optimizations implicitly using vector primitives.

However, all the existing programming models are based on traditional ISAs and require compile-time known vector lengths. Memory layout optimizations such as Hybrid SoA (Structure of Arrays) depend on slicing arrays based on vector lengths. In addition, higher-level models such as Sierra, build their own compiler support for easier programmability, loosing out on more powerful mainstream compilers.

As discussed in Chapter 2, we believe that programming applications for length-agnostic designs will require *both* programmer effort and compiler support. Hence, a programming model that captures programmer led algorithmic transformations while letting the compiler focus on auto-vectorization, could be an interesting research direction. However, we are not entirely certain how it would allow programmer-guided explicit vectorization hints while allowing

the compiler to vectorize an otherwise scalar-like code. We envision the following:

- Raising dynamic-vector length primitives to enable the programmer to express variable vector length.
- Express higher-level hints to aid vectorization analysis, possibly for outer-loop vectorization or specific instruction selection.

4.3 Multiple hardware prefetchers

Both Intel and ARM CPUs often contain multiple hardware prefetchers issuing prefetches concurrently based on the same memory access stream. This is done to cover a wider range of access patterns. However, since this can lead to excess traffic, these systems also come with throttling mechansims to improve accuracy.

Most hardware prefetchers proposals in academic research work, as discussed in Section 1.2.2, use a single prefetcher to maximize performance. Composite prefetcher models [49, 74, 18] use an ensemble of accurate sub-prefetchers for specific memory access patterns to improve overall prefetching accuracy and coverage. Since classification of memory access patterns are done at the hardware level, it faces two challenges: (1) requires microarchitectural changes to identify loops and pointer-based program semantics to detect access patterns; (2) on-chip tables to track per-PC classifications of streams can get costly for high instruction footprint datacenter workloads.

ProP, discussed in Chapter 3, can be expanded to detect multiple prefetch ac-

cess patterns in the profiling stage and use hints to guide a composite prefetcher model or select between multiple prefetchers (in Intel/ARM machines).

BIBLIOGRAPHY

[1] Arm cortex-a series programmer's guide for armv8-a: Virtual address tagging. `https://developer.arm.com/documentation/den0024/a/ch12s05s01`.

[2] Dennis Abts, Abdulla Bataineh, Steve Scott, Greg Faanes, Jim Schwarzmeier, Eric Lundberg, Tim Johnson, Mike Bye, and Gerald Schwoerer. The cray blackwidow: a highly scalable vector multiprocessor. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, 2007.

[3] Neil Adit and Adrian Sampson. Performance left on the table: an evaluation of compiler autovectorization for risc-v. *IEEE Micro*, 2022.

[4] Sam Ainsworth and Timothy M Jones. Software prefetching for indirect memory accesses. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2017.

[5] Arm. Arm neon. `https://developer.arm.com/Architectures/Neon`.

[6] 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*, 2019.

[7] Jean-Loup Baer and Tien-Fu Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, 1991.

[8] Sara S Baghsorkhi, Nalini Vasudevan, and Youfeng Wu. Flexvec: Auto-vectorization for irregular loops. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016.

[9] Thomas Ball and James R Larus. Branch prediction for free. *ACM SIGPLAN Notices*, 1993.

[10] Philip Bedoukian, Neil Adit, Edwin Peguero, and Adrian Sampson. Software-defined vector processing on manycore fabrics. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021.

[11] Eshan Bhatia, Gino Chacon, Seth Pugsley, Elvira Teran, Paul V Gratz, and Daniel A Jiménez. Perceptron-based prefetch filtering. In *Proceedings of the 46th International Symposium on Computer Architecture*, 2019.

[12] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, 2011.

[13] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Computer Architecture News*, 2011.

[14] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, 2003.

[15] Mark Buckler, Neil Adit, Yuwei Hu, Zhiru Zhang, and Adrian Sampson. Dense pruning of pointwise convolutions in the frequency domain. *arXiv preprint arXiv:2109.07707*, 2021.

[16] David Callahan, Jack J Dongarra, and David Levine. *Vectorizing compilers: A test suite and results*. Argonne National Laboratory, 1988.

[17] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. *ACM SIGARCH Computer Architecture News*, 1991.

[18] Gino Chacon, Elba Garza, Alexandra Jimborean, Alberto Ros, Paul V Gratz, Daniel A Jiménez, and Samira Mirbagher-Ajorpaz. Composite instruction prefetching. In *2022 IEEE 40th International Conference on Computer Design (ICCD)*, 2022.

[19] Tien-Fu Chen and Jean-Loup Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE transactions on computers*, 1995.

[20] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE journal of solid-state circuits*, 2016.

[21] Yuan Chou. Low-cost epoch-based correlation prefetching for commercial applications. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, 2007.

[22] I-Hsin Chung, Changhoan Kim, Hui-Fang Wen, and Guojing Cong. Application data prefetching on the ibm blue gene/q supercomputer. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012.

[23] Robert Cooksey, Stephan Jourdan, and Dirk Grunwald. A stateless, content-directed data prefetching mechanism. *ACM SIGPLAN Notices*, 2002.

[24] Scott Davidson, Shaolin Xie, Christopher Torng, Khalid Al-Hawai, Austin Rovinski, Tutu Ajayi, Luis Vega, Chun Zhao, Ritchie Zhao, Steve Dai, et al. The celerity open-source 511-core risc-v tiered accelerator fabric: Fast architectures and design methodologies for fast chips. *IEEE Micro*, 2018.

[25] Keith Diefendorff, Pradeep K Dubey, Ron Hochsprung, and HASH Scale. Altivec extension to powerpc accelerates media processing. *IEEE Micro*, 2000.

[26] Eiman Ebrahimi, Onur Mutlu, and Yale N Patt. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, 2009.

[27] Roger Espasa, Federico Ardanaz, Joel Emer, Stephen Felix, Julio Gago, Roger Gramunt, Isaac Hernandez, Toni Juan, Geoff Lowney, Matthew Mattina, et al. Tarantula: A vector extension to the alpha architecture. *ACM SIGARCH Computer Architecture News*, 2002.

[28] Roger Espasa, Mateo Valero, and James E Smith. Vector architectures: past, present and future. In *Proceedings of the 12th international conference on Supercomputing*, pages 425–432, 1998.

[29] Pierre Estérie, Joel Falcou, Mathias Gaunard, and Jean-Thierry Lapresté. Boost. simd: generic programming for portable simdization. In *Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing*, 2014.

[30] Timothée Ewart, Fabien Delalondre, and Felix Schürmann. Cyme: a li-

brary maximizing simd computation on user-defined containers. In *International Supercomputing Conference*, 2014.

[31] Keith I Farkas, Paul Chow, Norman P Jouppi, and Zvonko Vranesic. Memory-system design considerations for dynamically-scheduled processors. *ACM SIGARCH Computer Architecture News*, 1997.

[32] Joseph A Fisher, John R Ellis, John C Ruttenberg, and Alexandru Nicolau. Parallel processing: A smart compiler and a dumb machine. In *Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, 1984.

[33] Free Software Foundation. Auto-vectorization in gcc. `https://gcc.gnu.org/projects/tree-ssa/vectorization.html`.

[34] Free Software Foundation. Gcc. `https://gcc.gnu.org/`.

[35] J Gindele. Buffer block prefetching method. *IBM Technical Disclosure Bulletin*, 1977.

[36] RG Hintz. Control data star-100 processor design. In *COMPCOM*, 1972.

[37] Ibrahim Hur and Calvin Lin. Memory prefetching using adaptive stream detection. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, 2006.

[38] Intel. Intel sse4 programming reference. `https://www.intel.com/content/dam/develop/external/us/en/documents/18187-d9156103.pdf`, 2007.

[39] Intel. Intel® architecture instruction set extensions and future features. chapter 6. 2023.

[40] Yasuo Ishii, Mary Inaba, and Kei Hiraki. Access map pattern matching for high performance data cache prefetch. *Journal of Instruction-Level Parallelism*, 2011.

[41] Akanksha Jain and Calvin Lin. Linearizing irregular memory accesses for improved correlated prefetching. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 247–259, 2013.

[42] Akanksha Jain, Hannah Lin, Carlos Villavieja, Baris Kasikci, Chris Kennelly, Milad Hashemi, and Parthasarathy Ranganathan. Limoncello: Prefetchers for scale. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2024.

[43] Saba Jamilan, Tanvir Ahmed Khan, Grant Ayers, Baris Kasikci, and Heiner Litz. Apt-get: Profile-guided timely software prefetching. In *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022.

[44] Norman P Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *ACM SIGARCH Computer Architecture News*, 1990.

[45] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, 2017.

[46] 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*, 2015.

[47] 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. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022.

[48] Jinchun Kim, Seth H Pugsley, Paul V Gratz, AL Narasimha Reddy, Chris Wilkerson, and Zeshan Chishti. Path confidence based lookahead prefetching. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016.

[49] Sushant Kondguli and Michael Huang. Division of labor: A more effective approach to prefetching. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.

[50] Sanjeev Kumar and Christopher Wilkerson. Exploiting spatial locality in data caches using spatial footprints. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998.

[51] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2000.

[52] Nikita Lazarev, Neil Adit, Shaojie Xiang, Zhiru Zhang, and Christina Delimitrou. Dagger: Towards efficient rpcs in cloud microservices with near-memory reconfigurable nics. *IEEE Computer Architecture Letters*, 2020.

[53] Nikita Lazarev, Shaojie Xiang, Neil Adit, Zhiru Zhang, and Christina Delimitrou. Dagger: efficient and fast rpcs in cloud microservices with near-memory reconfigurable nics. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021.

[54] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. When prefetching works, when it doesn't, and why. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2012.

[55] Roland Leißa, Immanuel Haffner, and Sebastian Hack. Sierra: a simd extension for c++. In *Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing*, 2014.

[56] Wei-Fen Lin, Steven K Reinhardt, and Doug Burger. Reducing dram latencies with an integrated memory hierarchy design. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, 2001.

[57] LLVM. The llvm compiler infrastructure. `https://github.com/llvm/llvm-project/commit/e70533ae6c57756111689abf7826a3c632255866`, April, 2022.

[58] Chi-Keung Luk and Todd C Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, 1996.

[59] Scott Mahlke and Balas Natarajan. Compiler synthesized dynamic branch prediction. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*, 1996.

[60] Saeed Maleki, Yaoqing Gao, Maria J Garzar, Tommy Wong, David A Padua, et al. An evaluation of vectorizing compilers. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2011.

[61] Charith Mendis and Saman Amarasinghe. goslp: globally optimized superword level parallelism framework. *Proceedings of the ACM on Programming Languages*, 2018.

[62] Pierre Michaud. Best-offset hardware prefetching. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[63] Olga V Moldovanova and Mikhail G Kurnosov. Auto-vectorization of loops on intel 64 and intel xeon phi: Analysis and evaluation. In *International Conference on Parallel Computing Technologies*, pages 143–150. Springer, 2017.

[64] Todd Mowry and Anoop Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of parallel and Distributed Computing*, 1991.

[65] Todd C Mowry, Angela K Demke, Orran Krieger, et al. Automatic compiler-inserted i/o prefetching for out-of-core applications. In *OSDI*, 1996.

[66] Todd C Mowry, Monica S Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. *ACM Sigplan Notices*, 1992.

[67] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. A modern primer on processing in memory. In *Emerging Computing: From Devices to Systems: Looking Beyond Moore and Von Neumann*. Springer, 2022.

[68] Ramadass Nagarajan, Karthikeyan Sankaralingam, Doug Burger, and Stephen W Keckler. A design space evaluation of grid processor architectures. In *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*, 2001.

[69] Kyle J Nesbit and James E Smith. Data cache prefetching using a global history buffer. In *10th International Symposium on High Performance Computer Architecture (HPCA'04)*. IEEE, 2004.

[70] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. A compiler infrastructure for accelerator generators. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021.

[71] Dorit Nuzman, Ira Rosen, and Ayal Zaks. Auto-vectorization of interleaved data for simd. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[72] Dorit Nuzman and Ayal Zaks. Outer-loop vectorization: revisited for short simd architectures. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[73] NVIDIA. Nvidia cuda compiler driver nvcc. `https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/`.

[74] Samuel Pakalapati and Biswabandan Panda. Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[75] Subbarao Palacharla and Richard E Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the 21st annual international symposium on Computer architecture*, 1994.

[76] Biswabandan Panda. Clip: Load criticality based data prefetching for bandwidth-constrained many-core systems. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023.

[77] Alex Peleg and Uri Weiser. Mmx technology extension to the intel architecture. *IEEE micro*, 1996.

[78] Andrea Pellegrini. Arm neoverse n2: Arm's 2 nd generation high performance infrastructure cpus and system ips. In *2021 IEEE Hot Chips 33 Symposium (HCS)*, 2021.

[79] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. Using simpoint for accurate and efficient simulation. *ACM SIGMETRICS Performance Evaluation Review*, 2003.

[80] Karl Pettis and Robert C Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, 1990.

[81] Matt Pharr and William R Mark. ispc: A spmd compiler for high-performance cpu programming. In *2012 Innovative Parallel Computing (InPar)*, 2012.

[82] Leonardo Piga, Iyswarya Narayanan, Aditya Sundarrajan, Matt Skach, Qingyuan Deng, Biswadip Maity, Manoj Chakkaravarthy, Alison Huang, Abhishek Dhanotia, and Parth Malani. Expanding datacenter capacity with dvfs boosting: A safe and scalable deployment experience. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, 2024.

[83] Andrei Poenaru and Simon McIntosh-Smith. Evaluating the effectiveness of a vector-length-agnostic instruction set. In *European Conference on Parallel Processing*, 2020.

[84] Angela Pohl, Biagio Cosenza, Mauricio Alvarez Mesa, Chi Ching Chi, and Ben Juurlink. An evaluation of current simd programming models for c++. In *Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing*, pages 1–8, 2016.

[85] Angela Pohl, Mirko Greese, Biagio Cosenza, and Ben Juurlink. A performance analysis of vector length agnostic code. In *2019 International Conference on High Performance Computing & Simulation (HPCS)*, 2019.

[86] Vasileios Porpodas, Alberto Magni, and Timothy M Jones. Pslp: Padded slp automatic vectorization. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2015.

[87] Vasileios Porpodas, Rodrigo CO Rocha, and Luís FW Góes. Vw-slp: auto-vectorization with adaptive vector width. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, 2018.

[88] Seth H Pugsley, Zeshan Chishti, Chris Wilkerson, Peng-fei Chuang, Robert L Scott, Aamer Jaleel, Shih-Lien Lu, Kingsum Chow, and Rajeev Balasubramonian. Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[89] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 2013.

[90] Mahesh Rajan, Douglas W Doerfler, Mike Tupek, and Simon Hammond. An investigation of compiler vectorization on current and next-generation intel processors using benchmarks and sandia's sierra applications. 2015.

[91] Cristóbal Ramírez, César Alejandro Hernández, Oscar Palomar, Osman Unsal, Marco Antonio Ramírez, and Adrián Cristal. A risc-v simulator and benchmark suite for designing and evaluating vector architectures. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2020.

[92] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE micro*, 2010.

[93] RISC-V 2021. Openmp application programming interface. `https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf`, 2018.

[94] RISC-V 2021. Working draft of the proposed risc-v v vector extension. `https://github.com/riscv/riscv-v-spec`, 2021.

[95] Arch D Robison. Composable parallel patterns with intel cilk plus. *Computing in Science & Engineering*, 2013.

[96] Richard M Russell. The cray-1 computer system. *Communications of the ACM*, 1978.

[97] Richard M Russell. The cray-1 computer system. *Communications of the ACM*, 1978.

[98] Suleyman Sair, Timothy Sherwood, and Brad Calder. A decoupled predictor-directed stream prefetching architecture. *IEEE Transactions on Computers*, 2003.

[99] Jennifer B Sartor, Subramaniam Venkiteswaran, Kathryn S McKinley, and Zhenlin Wang. Cooperative caching with keep-me and evict-me. In *9th Annual Workshop on Interaction between Compilers and Computer Architectures (INTERACT'05)*, 2005.

[100] Manjunath Shevgoor, Sahil Koladiya, Rajeev Balasubramonian, Chris Wilkerson, Seth H Pugsley, and Zeshan Chishti. Efficiently prefetching complex address patterns. In *Proceedings of the 48th International Symposium on Microarchitecture*, 2015.

[101] Sergi Siso, Wes Armour, and Jeyarajan Thiyagalingam. Evaluating auto-vectorizing compilers through objective withdrawal of useful information. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2019.

[102] Alan Jay Smith. Sequential program prefetching in memory hierarchies. *Computer*, 1978.

[103] Alan Jay Smith. Cache memories. *ACM Computing Surveys (CSUR)*, 1982.

[104] Eric Smith. Memory bandwidth per core and per socket for intel xeon and amd epyc. https://www.servethehome.com/memory-bandwidth-per-core-and-per-socket-for-intel-xeon-and-amd-

[105] Stephen Somogyi, Thomas F Wenisch, Anastasia Ailamaki, and Babak Falsafi. Spatio-temporal memory streaming. *ACM SIGARCH Computer Architecture News*, 2009.

[106] Stephen Somogyi, Thomas F Wenisch, Anastassia Ailamaki, Babak Falsafi, and Andreas Moshovos. Spatial memory streaming. *ACM SIGARCH Computer Architecture News*, 2006.

[107] 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*, 2007.

[108] Akshitha Sriraman and Abhishek Dhanotia. Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 733–750, 2020.

[109] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, et al. The arm scalable vector extension. *IEEE Micro*, 2017.

[110] Majedul Haque Sujon, R Clint Whaley, and Qing Yi. Vectorization past dependent branches through speculation. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, 2013.

[111] Tuan Ta, Khalid Al-Hawaj, Nick Cebry, Yanghui Ou, Eric Hall, Courtney Golden, and Christopher Batten. Big. vlittle: On-demand data-parallel acceleration for mobile systems on chip. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022.

[112] Alexa VanHattum, Rachit Nigam, Vincent T Lee, James Bornholt, and Adrian Sampson. Vectorization for digital signal processors via equality saturation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021.

[113] Zhenlin Wang, Doug Burger, Kathryn S McKinley, Steven K Reinhardt, and Charles C Weems. Guided region prefetching: A cooperative hardware/software approach. *ACM SIGARCH Computer Architecture News*, 2003.

[114] Zhenlin Wang, Kathryn S McKinley, Arnold L Rosenberg, and Charles C Weems. Using the compiler to improve cache replacement decisions. In *Proceedings. International Conference on Parallel Architectures and Compilation Techniques*, 2002.

[115] WJ Watson. The ti asc: a highly modular and flexible super computer architecture. In *Proceedings of the December 5-7, 1972, fall joint computer conference, part I*, 1972.

[116] Shlomo Weiss and James E Smith. A study of scalar compilation techniques for pipelined supercomputers. *ACM SIGARCH Computer Architecture News*, 1987.

[117] 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*, 2008.

[118] Thomas F Wenisch, Michael Ferdman, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. Making address-correlated prefetching practical. *IEEE micro*, 2010.

[119] XLA. Xla (accelerated linear algebra). `https://github.com/openxla/xla`.

[120] Zhongcheng Zhang, Yan Ou, Ying Liu, Chenxi Wang, Yongbin Zhou, Xiaoyu Wang, Yuyang Zhang, Yucheng Ouyang, Jiahao Shan, Ying Wang, et al. Occamy: Elastically sharing a simd co-processor across multiple cpu cores. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023.

ProQuest Number: 31556872

INFORMATION TO ALL USERS
The quality and completeness of this reproduction is dependent on the quality and completeness of the copy made available to ProQuest.

Distributed by
ProQuest LLC a part of Clarivate (2025).
Copyright of the Dissertation is held by the Author unless otherwise noted.

This work is protected against unauthorized copying under Title 17, United States Code and other applicable copyright laws.

This work may be used in accordance with the terms of the Creative Commons license or other rights statement, as indicated in the copyright statement or in the metadata associated with this work. Unless otherwise specified in the copyright statement or the metadata, all rights are reserved by the copyright holder.

ProQuest LLC
789 East Eisenhower Parkway
Ann Arbor, MI 48108 USA