Composite Instruction Prefetching

Gino Chacon*† ginochacon@tamu.edu
Elba Garza*¶ elba@cs.washington.edu
Alexandra Jimborean§ alexandra.jimborean@um.es
Alberto Ros§ aros@ditec.um.es
Paul V. Gratz†‡ pgratz@gratz1.com
Daniel A. Jiménez‡ djimenez@acm.org
Samira Mirbagher-Ajorpaz‖ smirbag@ncsu.edu

*Co-first Authors
†Department of Electrical and Computer Engineering, Texas A&M University, College Station, Texas, USA
‡Department of Computer Science and Engineering, Texas A&M University, College Station, Texas, USA
§Computer Engineering Department, University of Murcia, Murcia, Spain
¶Paul G. Allen School of Computer Science, University of Washington, Seattle, Washington, USA
‖Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, North Carolina, USA

***Abstract*—Prefetching is a pivotal mechanism for effectively masking latencies due to the processor/memory performance gap. Instruction prefetchers prevent costly instruction fetch stalls by requesting blocks of instruction memory in advance of their use to keep the pipeline front-end busy. the rapidly increasing instruction footprints of modern workloads have amplified the importance of such research.**

We propose a framework to leverage the complementary prefetching behaviors of existing prefetching techniques to create composite prefetchers. We show that recently proposed instruction prefetching techniques leverage different mechanisms from one another and find that in many cases, different prefetchers are complementary to each other. Composite prefetching allows for higher performance at lower storage overheads by combining the coverage of different complex prefetchers. We demonstrate a framework for selecting and combining state-of-the-art complex prefetchers, in a "plug-and-play" fashion, to identify the best performing combinations at various hardware overheads. We show that for every storage capacity constraint analyzed, composite prefetching outperforms prior prefetching schemes with greater improvements shown at smaller capacity constraints.

***Index Terms*—instruction prefetching; hardware prefetching; first-level caches; datacenter applications**

I. Introduction

As server and data center-based computing increases, processor microarchitecture must adapt to their workloads' challenging behaviors. Server-based computing services are characterized by deep software stacks, where even simple user requests traverse multiple software layers, touching megabytes of code in the process [1]. The instruction footprints of these server workloads can quickly overburden L1 instruction (L1-I) caches resulting in significant critical-path stalls. Recent studies have found that large footprints are endemic to modern server workloads and will only grow larger over time—at a rate of 20% per year—as software stacks deepen to accommodate more complex applications [2]. This overwhelming use of front-end resources is a well-known phenomenon dubbed the *front-end bottleneck* [1]. The front-end bottleneck challenges hardware architects as programs' long-term behaviors can no longer be fully mapped in the L1-I cache nor captured by branch prediction structures [3]. Thus, microarchitects must limit the effects of the front-end bottleneck while keeping to strict timing, area, and power constraints.

One tool for addressing the front-end bottleneck is prefetching, a technique that can effectively mask memory access latencies by speculating on future memory reference streams, to reduce costly cache misses [4], [5]. Prefetching is a viable mechanism for reducing misses in both instruction and data caches. Prefetching requires predicting upcoming memory accesses and issuing preemptive memory requests before explicit requests by memory are necessary [6], [4], [7], [8], [9].

We find that, due to their independent implementations, recently proposed instruction prefetchers [10], [11], [12], [13], [14], [15], [16], [17] often behave differently dependent on workload and program phase, leading to different prefetch suggestions at different times, and thus they vary in terms of timeliness, coverage, and accuracy. Interestingly, we also find that as the hardware budget of each prefetcher is reduced to make the prefetcher more practical in a production environment, they behave more distinctively and complementarily. This finding argues for the benefits of leveraging multiple prefetchers *in combination*, particularly when implemented in reasonable, buildable hardware budgets.

While prior work has meticulously integrated simple prefetchers to create composite prefetchers for both instruction [18] and data [9], [19], such component-based composite prefetchers do not allow for *interchangeability of components* and must be tuned extensively to ensure each component captures specific application behaviors. This requires in-depth knowledge of the prefetching behavior of each component, and an idea of how they may complement each other regarding coverage and precision. These works also require a mechanism to identify whether recent accesses fit a particular pattern

attributed to a specific component, increasing the likelihood of the prefetcher misidentifying the application's behavior. By contrast, this work seeks to coordinate and interchange multiple complex instruction prefetchers, allowing for broader coverage to overcome the timing constraints of traditional composite prefetchers.

Based on the increasing instruction footprint size of modern server workloads and the success of individual prefetchers, we propose a new approach to composite prefetching that *requires no knowledge of the component prefetchers* and allows for interchangeable components to enable prefetching design space exploration. Our approach integrates preexisting complex prefetchers as components within a single composite prefetcher. Each prefetcher design may capture different instruction streams, and their combined prefetch behavior results in higher coverage and performance than an individual prefetcher at an equivalent size. Aside from varying the metadata storage required by each prefetcher, each component is considered a black box. This approach mitigates the burden of creating tailor-made components to capture specific application behavior when creating a composite prefetcher. As a practical design methodology, our approach would enable industry to easily interchange any desired prefetcher from the academic literature and study them in combination without spending valuable design time evaluating each component's advantages or shortcomings. In summary, our contributions in this work are as follows:

- We characterize prior work, a selection of complex instruction prefetchers, to understand their complementary nature and composability at differing sizes.
- We study the feasibility of combining a set of state-of-the-art hardware prefetchers to better capture instruction stream behavior.
- We identify a simple hybridization scheme for combining existing prefetchers to leverage their complementary behavior. Our scheme can integrate multiple complex prefetchers with no prior knowledge of function.
- We perform a full design-space exploration for the set of hardware prefetchers to identify their best performing combinations at various hardware budgets.
- Using our scheme, we demonstrate that a combination of multiple state-of-the-art prefetchers can outperform its components at the same hardware budget.

II. Background and Motivation

This section provides a background on the component prefetchers we consider for generating composite prefetchers, and our motivation for exploring composite prefetching as a solution to the front-end bottleneck.

A. Modern Instruction Prefetchers

Recently proposed instruction prefetchers speculate on future references using varying underlying mechanisms. Due to their differences, they each tend to be more or less efficient at predicting future references for particular workloads and program phases. Thus, by combining these existing prefetchers, we can create a single unified prefetcher better than the sum of its parts. Here we examine several recently proposed prefetchers and classify them based on function.

Each class of prefetcher operates based on specific principles surrounding an application's behavior to predict future instructions. Recently proposed prefetchers can be classified based on how they train, represent instruction stream behavior, and select prefetch candidates. This is similar to recent classifications of data prefetchers [20]. Recent work we consider for our composite prefetchers falls into the following classes of prefetchers:

1) Control-Flow-Graph Recreation: These prefetchers recreate an application's control-flow graph (CFG). Nodes represent basic blocks within a graph, with edges representing control-flow (branch) instructions. The prefetcher uses the address of L1-I accesses to find a starting point to traverse the CFG to find prefetch candidates. Confidence is generally assigned based on observations of the control flow to indicate the application's likelihood to take a particular execution path.

Barça: The Branch Agnostic Region Searching Algorithm, or Barça [12], creates a control-flow graph to map regions of instruction blocks and their relative control flow.

PIPS: Prefetching Instructions with Probabilistic Scouts, or PIPS [17] recreates an application's CFG using a Line-History Table to connect cache lines recently accessed together, tracking the probability of traversing a particular edge.

2) Temporal Prefetchers: These prefetchers attempt to predict the future instruction stream by identifying accesses that cause cache misses, recording the following misses, and replaying them when a triggering access is seen. This style of prefetcher emphasizes timeliness by prefetching misses in an instruction stream well before the front-end requests them. We classify the following prefetchers as temporal prefetchers:

EIP: The Entangling Instruction Prefetcher [10], [21] "entangles" instructions together to provide prefetch timelines, accounting for the prefetch latency to identify the suitable instruction to trigger the prefetch.

FNL-MMA: FNL-MMA [13] combines a Footprint Next Line Prefetcher (FNL) to predict the "not so distant" future, while the Multiple Miss Ahead Predictor (MMA) takes advantage of predictable cache miss sequences.

TAP: The Temporal Ancestry Prefetcher [14] augments a next-line prefetcher with temporal-based histories leading to a program counter (PC) based on the observation that most cache lines are not rereferenced once they fill the cache.

MANA: MANA [15] creates spatial regions in a set-associative table, tracking a triggering address and a footprint to indicate which blocks within a region are accessed. MANA traverses its table when prefetching, loading a stream address buffer with prefetch candidates. searched to a certain depth.

3) Branch-Oriented Prefetchers: Unlike CFG-based prefetchers, these prefetchers do not recreate the CFG but rather use branch-related information to make predictions about future accesses and cover branch targets. This information includes the branch-target-buffer, the return-address-stack (RAS), or other branch structures. We use the following branch-oriented prefetchers in our work:

D-JOLT: D-JOLT [11] consists of multiple simple prefetchers of varying characteristics. It uses a long-range prefetcher to cover the distant future with higher coverage, a short-range prefetcher to cover the near future with higher accuracy, and a "fall-back" prefetcher.

JIP: JIP [16] is composed of multiple prefetchers that target specific instruction stream behavior, such as sequential accesses within basic-blocks, branches with a single target, and branches to multiple targets.

B. Complementary Prefetchers

While some prefetchers have high performance at lower hardware budgets, they cannot leverage more storage to achieve higher performance. Each prefetcher's performance varies and operates on different principles, which can be broadly classified (Sec. IV-B), but their classification does not predict their performance at various hardware budgets. Figure 1 illustrates this by showing the overlap of unique addresses targeted by each prefetcher at sizes of 10KB and 128KB. Ideally, complementary prefetchers have lower overlap to facilitate different instruction stream behaviors. Lower hardware budgets limit the misses each prefetcher learns and targets, resulting in a low overlap between most prefetchers except for FNL+MMA and JIP, PIPS, and TAP. At higher storage budgets, the prefetchers are less constrained, capture more misses, and thus converge towards similar behavior. *This indicates that at different hardware budgets, a combination of prefetchers have vastly different prefetching behavior and potentially capture different instruction streams.* However, if prefetch streams are too dissimilar, there is a potential for destructive interference between the prefetchers as they could each aggressively prefetch different instruction streams, resulting in a high amount of thrashing in the already encumbered L1-I. To this end, we propose a composite prefetching framework and methodology for searching for the best performing combination of prefetchers at various hardware budgets.

C. Composite Prefetching

Few composite prefetchers exist in the prior work, mainly in the data prefetching domain. Division of Labor, or DOL [9], attempts to exploit both simple and complex access patterns using a collaboration of specialized subcomponents for each pattern. DOL is extendable with additional components as more access patterns are identified. Note that the hardware designer must identify missing or necessary access patterns, making DOL limited by the designer's knowledge.

Bouquet of Instruction Pointers [19] creates a composite L1 data prefetcher that uses a "bouquet" of pointers to classify instruction pointers and issue data requests based on the classification. This technique covers and identifies a handful of memory access patterns that drive prefetches.

The above works focus on data prefetching that relates specific instructions to data they access. While instruction prefetching and data prefetching are similar, as they attempt to hide access latencies, their access patterns and relationships to data diverge. Instruction prefetchers target instructions themselves, causing control flow to be an important factor. Divide and Conquer Frontend Bottleneck [18] warns against BTB-directed instruction prefetches, presenting the "harmful effects" of making instruction prefetchers dependent on BTB content. Instead, it proposes dividing the front-end bottleneck into a sequential prefetcher to cover sequential misses, a discontinuity prefetcher, and pre-decoding prefetch blocks to reduce BTB misses. This divide-and-conquer method has the same area overhead as a BTB-directed prefetcher but outperforms it by 5% on average for their selected workloads.

As seen by the works described above, the concept of combining prefetchers, both in data and instruction prefetching, is not novel in itself. However, these component prefetchers are *non-interchangeable* and tuned for hardware size and prefetch specialty by the designer, requiring in-depth knowledge of each component.In contrast, our proposition requires no knowledge of the prefetcher components and allows for previously unexplored component interchangeability.

	Barca	D-JOLT	FNL-MMA	EIP	JIP	PIPS	TAP	Mana
Barca	100.0%	33.5%	34.2%	34.1%	34.2%	34.2%	34.2%	34.2%
D-JOLT	33.5%	100.0%	42.9%	42.4%	42.9%	42.9%	42.8%	42.7%
FNL-MMA	34.2%	42.9%	100.0%	48.6%	69.2%	68.8%	66.8%	51.5%
EIP	34.1%	42.4%	48.6%	100.0%	48.6%	48.6%	48.6%	47.5%
JIP	34.2%	42.9%	69.2%	48.6%	100.0%	74.4%	78.5%	51.6%
PIPS	34.2%	42.9%	68.8%	48.6%	74.4%	100.0%	71.7%	51.6%
TAP	34.2%	42.8%	66.8%	48.6%	78.5%	71.7%	100.0%	51.4%
Mana	34.2%	42.7%	51.5%	47.5%	51.6%	51.6%	51.4%	100.0%

(a) Percent overlap of unique prefetch targets between two prefetchers sized at roughly 10KB each for a subset of CVP traces.

	Barca	D-JOLT	FNL-MMA	EIP	JIP	PIPS	TAP	Mana
Barca	100.0%	62.7%	62.7%	62.6%	63.2%	63.2%	63.0%	62.6%
D-JOLT	62.7%	100.0%	66.9%	68.1%	71.0%	70.9%	69.9%	69.1%
FNL-MMA	62.7%	66.9%	100.0%	67.1%	68.8%	68.8%	67.9%	67.4%
EIP	62.6%	68.1%	67.1%	100.0%	71.7%	71.7%	70.3%	70.0%
JIP	63.2%	71.0%	68.8%	71.7%	100.0%	99.6%	77.8%	77.2%
PIPS	63.2%	70.9%	68.8%	71.7%	99.6%	100.0%	77.8%	77.2%
TAP	63.0%	69.9%	67.9%	70.3%	77.8%	77.8%	100.0%	73.3%
Mana	62.6%	69.1%	67.4%	70.0%	77.2%	77.2%	73.3%	100.0%

(b) Percent overlap of unique prefetch targets between two prefetchers sized at roughly 128KB each for a subset of CVP traces.

Fig. 1: The overlap between individual prefetchers at the smallest (10KB) and largest (128KB) hardware budgets. The higher the percentage, the more the two prefetchers overlap.

III. Design and Implementation

This section describes our proposed design of a composite prefetcher, consisting of two or more prefetchers. We begin by

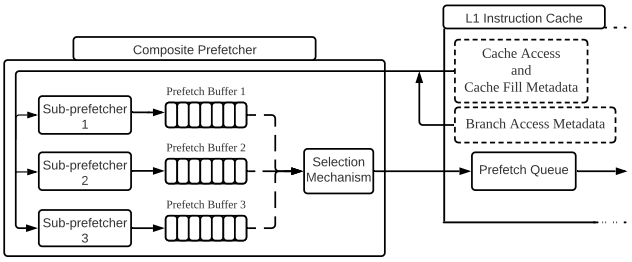

Fig. 2: Overview of the composite prefetcher's organization.

describing the hardware framework that enables the integration of multiple prefetchers. We then discuss how the composite prefetcher generates and issues prefetch candidates to the L1-I.

A. Composite Prefetcher Organization

A key design constraint of our composite prefetcher is that it should be comparable to a single prefetcher using the same hardware budget. Thus, we scale down the budget used in the various prefetchers described in Section II to use multiple prefetchers with a comparable total budget of a single prefetcher. For instance, integrating two 10KB prefetchers will have comparable hardware overhead as a single 20KB prefetcher. A goal when integrating prefetcher components is for their individual operation to remain intact. Thus, other than modifying their structures to reduce their metadata state storage, we make no further changes to any prefetcher component.

In this work, we explore integrating mixes of two to four prefetchers. Figure 2 illustrates the hardware framework for prefetchers. Each complex prefetcher component, or ***subprefetcher***, receives metadata from the L1-I cache regarding cache accesses, misses, and fills. Following the subprefetchers, a ***subprefetcher buffer (SPFB)*** is available to each prefetcher that is the same size as the L1-I prefetch queue. This allows the prefetchers to operate as they would individually without being affected by each other's prefetch queue's bandwidth pressure. Finally, a selection mechanism is placed at the head of the buffers to select a prefetcher's buffer based on a selection policy. Beyond the state required per prefetcher, this buffering and selection mechanism requires 32 entries per SPFB to hold 56-bit prefetch cache line addresses (assuming 64B cache lines), resulting in a 224B overhead per subprefetcher. Further discussion on the operation of the composite prefetcher is provided below in Section III-C.

An essential property of a composite prefetcher organization is that although the subprefetchers are technically not aware of each other when run in tandem, they adapt to each others' behavior through the misses that are covered versus uncovered during execution. Since the majority of prefetchers we consider are trained only on cache misses, once one prefetcher learns to cover a given miss, other subprefetchers can quickly adapt to disregard the metadata state needed to cover that miss since it is no longer considered a miss from their perspective. As we will show, this is desirable since this allows subprefetchers to use their limited storage to focus on the misses not covered by the accompanying subprefetchers.

B. L1-I Cache Metadata and Subprefetcher Training

We provide each prefetcher information on cache demand accesses, prefetch hits, branch information and results, and the effects of cache fills, such as the filling cache address and the victim cache line's address. Each prefetcher is treated as a "black box" with regards to the metadata it receives and is not tuned to cover specific instruction stream behavior, as opposed to prior composite prefetcher work [18], [9], [19]. Each subprefetcher trains based on their individual training policy and may disregard any provided information.

C. Composite Prefetcher Operation

Each prefetcher generates a set of prefetch candidates on a cache access that it places in the SPFBs. Each cycle following generation, the prefetch selection mechanism transfers the head of a particular SPFB into the L1-I cache's prefetch queue using a round-robin selection mechanism. We explored other selection mechanisms but found that round-robin was sufficient because much of the time, only one prefetcher is filling its SPFB while the other SPFBs are empty. If the prefetch queue is full or there is no available Miss Status Handling Register (MSHR), the selection mechanism does not continue to move prefetches into the queue. When generating a new stream of prefetches, the head of the SPFB is set to the first free buffer entry, and new prefetches fill the buffer. Newly generated prefetches overwrite the current contents of the SPFB if it is full, removing stale prefetches from the previous generation that could pollute the L1-I.

IV. Evaluation

This section describes the design space exploration and evaluation of a composite prefetcher. We begin by describing our simulation environment and evaluation methodology. Next, we evaluate composite prefetching schemes composed of two, three, and four subprefetchers and the design space surrounding subprefetcher selection at different hardware budgets. We then describe the evaluation of individual subprefetchers' contribution to performance. Finally, we discuss the best performing combination of subprefetchers at hardware budgets of 20KB, 30KB, 40KB, 64KB, and 128KB.

A. Methodology

We perform design-space exploration and evaluation of the composite-prefetcher framework using the ChampSim simulator [22]. Featuring an aggressive front-end similar to Fetch-Directed Prefetching [23], [24] and models a Branch Target Buffer (BTB) that includes an indirect BTB and return address stack. We configure the simulator to reflect recent Intel's Sunny Cove microarchitecture with the parameters in Table I.

TABLE I: Simulated Baseline System Configuration

Processor Configuration	
Clock Frequency	4GHz
Fetch Queue	64 entries
Decode Queue	32 entries
Dispatch Queue	32 entries
Reorder Buffer	352 entries
Load Queue	128 entries
Store Queue	72 entries
Fetch width	6 instructions
Decode width	6 instructions
Dispatch width	6 instructions
Memory Configurations	
L1 I-Cache	32KB, 8 ways, 64 sets, no prefetcher
L1 D-Cache	48KB, 12 ways, 64 sets, next line prefetcher
L2 Cache	512KB, 8 ways, 1024 sets, spp
LLC Cache	2MB, 16 ways, 2048 sets

TABLE II: Best performing prefetcher combinations for Composite-2 for hardware budgets of 20KB, 30KB, 40KB, 64KB, and 128KB divided evenly between subprefetchers.

Hardware Budget	Subprefetcher-1	Subprefetcher-2
20KB	Barça	FNL+MMA
30KB	FNL+MMA	MANA
40KB	FNL+MMA	EIP-ISCA
64KB	D-JOLT	FNL+MMA
128KB	FNL+MMA	EIP-ISCA

For our evaluation, we employ a subset of the traces from the 1st Championship Value Prediction (CVP-1) [25], provided by Qualcomm Datacenter Technologies and ported to the ChampSim format. We selected CVP traces that showed at least one MPKI (miss per kilo-instruction) at the L1-I and L2C in our baseline configuration and demonstrated high performance potential beyond a next-line L1-I prefetcher with reference to the maximum performance as measured by an Oracle L1-I prefetcher. The selected CVP traces demonstrate MPKIs ranging from 3 to 48 at the L1-I cache. All benchmarks maintain low MPKIs in the L1-D, indicating that L1-I miss limits these workloads' performance. As many recent works point to the instruction cache misses becoming more critical in cloud and server workloads, we chose this subset to represent emerging, high instruction cache pressure applications. Each benchmark shown is executed for 50M instructions to warm up the predictors and caches, with another 50M instructions executed to measure performance. Despite the short simulation lengths, we emphasize that these traces demonstrate high L1-I miss rates analogous to the high miss rates seen in modern datacenter workloads.

B. Hardware Constraints and Instruction Prefetcher Performance

Existing academic instruction prefetchers have significant coverage when their metadata storage state is unconstrained. However, they struggle when implemented with more realistic

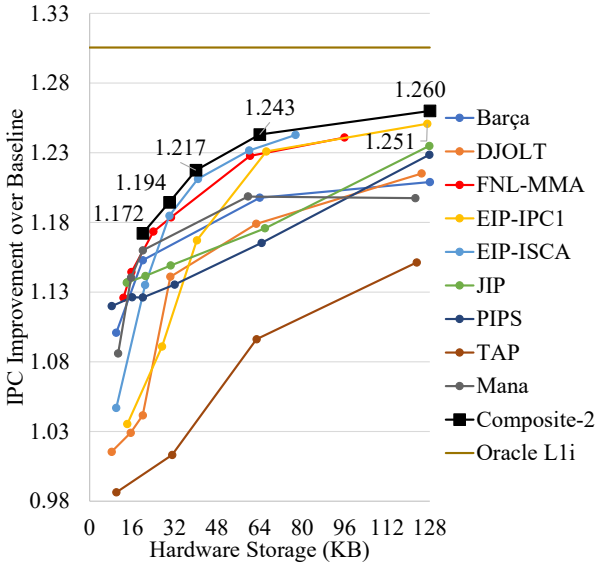

Fig. 3: Prefetcher performance versus hardware budget, including the best performing combined prefetcher at each hardware budget. At each budget, the composite prefetcher outperforms not only its subprefetcher components but the best performing single-prefetcher.

amounts of storage than industrial designs expect. Figure 3 shows the individual performance of the prefetchers described in Section II-A as the storage budget for the prefetcher increases. We compare each prefetcher's performance against a baseline system with no instruction prefetching, an SPP [26] data prefetcher in the L2 cache, and a least-recently-used (LRU) replacement policy used in all cache levels.

We include an Oracle instruction prefetcher that covers all non-compulsory L1-I misses. The Oracle also fills the unified L2 and L3 caches to mimic the data interference an L1-I prefetch stream would have. The oracle represents a reasonably tight upper bound on the attainable performance from instruction prefetching. As Figure 3 shows, at the maximum hardware budget evaluated, EIP comes within 5% of Oracle's performance. As expected, all prefetchers suffer lower performance benefits at lower hardware storage budgets and enjoy increased performance benefits as the hardware budget increases. Performance tends to fall significantly at hardware budgets of 64KB and less, and several prefetchers take turns showing the best performance at different points in the space. In general, we observe three specific trends in Figure 3 based on the prefetchers' performance at various hardware budgets: low-budget friendly, budget-sensitive, or budget indifferent.

1) Budget Sensitive Prefetchers: These prefetchers (i.e., EIP, D-JOLT, and TAP) experience heavy performance degradation at lower hardware budgets, with performance substantially increasing with the hardware budget. In particular, EIP sees ~20% performance improvement from increasing the hardware budget from 15KB to 128KB. These prefetchers are

ideal for high-budget designs but may not be reasonable for smaller designs.

2) Low-Budget Friendly Prefetchers: These prefetchers experience a drop in performance at lower hardware budgets while still exhibiting high performance gains from increased hardware budgets. Barça, FNL+MMA, JIP, and PIPS follow this trend, seeing moderate performance benefits at sizes of 10-30KB and scaling as the hardware budget increases. Though their improvements from increasing hardware budgets are not as drastic as EIP, these prefetchers provide consistent performance benefits as their design scales, indicating they are viable options for low-budget *and* high-budget designs.

3) Budget Indifferent Prefetchers: This trend is observed when prefetchers perform well at lower hardware budgets, but only experience modest benefits from an increase in hardware budget. MANA follows this trend, being less affected by a lower hardware budget of 10-15KB. This prefetcher is resilient to lower storage but does not benefit significantly from an increased hardware budget, seeing only a 5% performance benefits from an increased budget of 15KB to 128KB, making it better suited for low hardware budget designs with consistent performance as the prefetcher's hardware budget scales.

C. Selecting Composite Prefetcher Subprefetchers

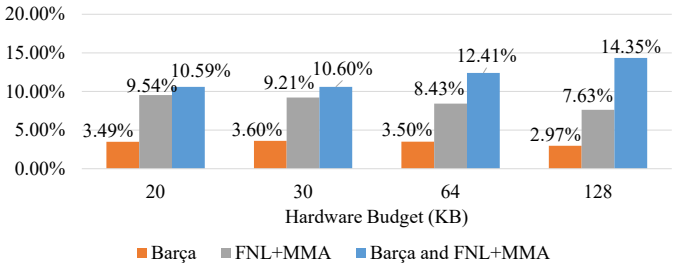

Fig. 4: Miss coverage, in a Composite-2 of Barça & FNL+MMA covered by Barça, FNL+MMA or both at various hardware budgets

The design space of exploring an N-composite scheme, with the possibility of 8 different prefetchers filling any one slot within an N-composite scheme, with prefetchers of various hardware budgets to meet an overall hardware budget is exceedingly large. Given 8 possible subprefetchers of various sizes, the design space increases exponentially as the overall hardware budget increases. For our experiments, each subprefetcher's size is determined by the overall hardware budget divided evenly between the subprefetchers. For example, a 30KB composite prefetcher may be composed of two 15KB prefetchers or three 10KB prefetchers. The results of Figure 3 direct the design exploration for the composite design based on a prefetcher's performance relative to its hardware budget.

For each hardware budget, we perform a design space exploration for composite prefetchers composed of two, three, and four subprefetchers. Our design space exploration combines N

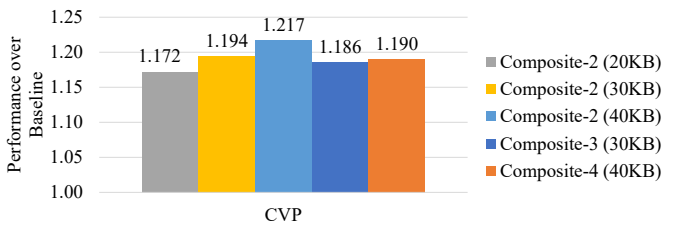

Fig. 5: Performance comparison of best composites of 2, 3 and 4 prefetchers for different metadata storage state.

of the 8 possible prefetchers and then evaluates the resultant composite prefetcher. Each N-composite prefetcher has a range of performance results, with the best performing combination varying based on the hardware budget. Figure 5 shows the performance of the best performing composite of 2, 3 and 4 prefetchers at 20KB, 30KB and 40KB.

Here, the hardware budget of the composites is divided evenly between each subprefetchers, i.e. for Composite-2 at 20KB, each subprefetcher has roughly 10KB of state. In the figure, we see that composites of 2 significantly outperform composites of 3 and composites of 4 prefetchers. Even the smaller 20KB Composite-2 prefetcher approaches the performance of the Composite-3 and Composite-4 prefetchers within ~2%. The allocated budget per prefetcher is reduced as the number of prefetchers increases to keep the overall budget within limits. Thus, the effectiveness of each prefetcher diminishes. We conclude that composing a high number of prefetchers with a reasonable budget is not promising. As a result, we focus on Composite-2 prefetchers for the remainder of this work.

Table II lists the subprefetchers from the best performing composite-2 prefetcher found in our design space exploration at each storage size. Interestingly, the best performing composite changes significantly at each hardware budget.

D. Full results comparison

Figure 3 shows the results of all prefetchers examined, scaled to different sizes, along with the best performing Composite-2 prefetcher combination discussed in Section IV-C and listed in Table II. In general, the Composite-2 prefetcher outperforms all single prefetchers at every metadata storage size, often by significant margins. Composite-2's performance increases as its subprefetchers' can capture more of the workloads' behavior but see the highest performance benefits at low hardware budgets.

Figure 4 shows the misses that Barça and FNL+MMA cover individually and the overlap in their access coverage when combined at varying hardware budgets in a Composite-2 prefetcher. As Composite-2 is scaled, the coverage overlap between the two prefetchers increases. Interestingly, for small budgets, each subprefetcher covers a greater fraction of misses than both cover together. This illustrates that at smaller bud-

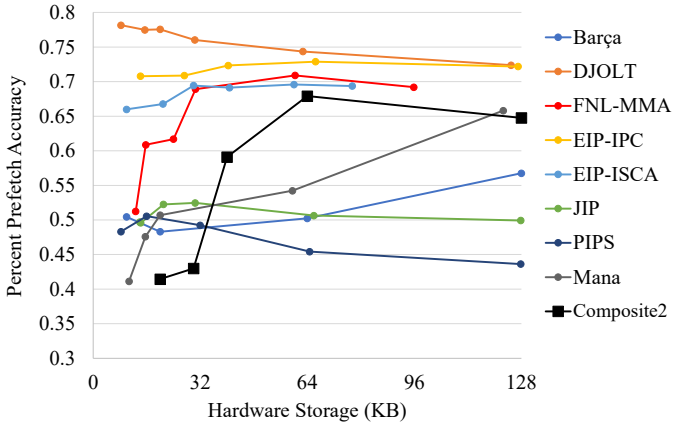

Fig. 6: Accuracy vs. Hardware Storage (KB) at each metadata storage point for all prefetchers and Composite-2.

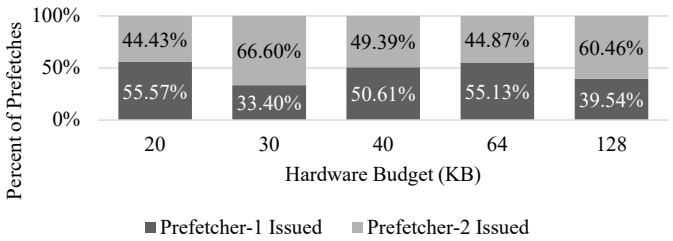

Fig. 7: Percentage of prefetches issued from each component prefetcher in the best performing Composite-2 prefetcher at each storage overhead. Generally, one subprefetcher does not tend to dominate the prefetches Composite-2 produces.

gets, each prefetcher tends to focus on misses it is better able to cover, yielding greater metadata storage efficiency than can be achieved by a single prefetcher at even double the size.

E. Subprefetcher Behavior

1) Accuracy and Issued Prefetches: We evaluate the accuracy of a composite-2 prefetcher compared to the potential subprefetcher components in Figure 6. Composite-2's accuracy depends on its component subprefetchers' behavior, with its accuracy increasing as the hardware budget increases. Each prefetcher, except DJOLT, does not train off cache accesses that hit on a prefetched line, resulting in each prefetcher training on misses not covered by other prefetchers. This increases the orthogonality between prefetchers' predictions, as discussed in section IV-E2, while decreasing overall accuracy.

The round-robin selection mechanism considers each subprefetcher's prefetch stream. A more complex selection mechanism can be implemented to prioritize prefetches from a particular subprefetcher. However, the selection mechanism's impact relies on multiple subprefetchers generating predictions simultaneously. Our evaluation finds that only one subprefetcher generates prefetches for 80% of demand cache accesses at any hardware budget, indicating complex selection mechanisms are unlikely to identify prefetching opportunities that would benefit from prioritizing one subprefetcher over another. Figure 7 shows the prefetches issued by each subprefetcher in Composite-2. Overall, no subprefetcher dominates the prefetch contributions, allowing each subprefetcher to provide complementary prefetch candidates.

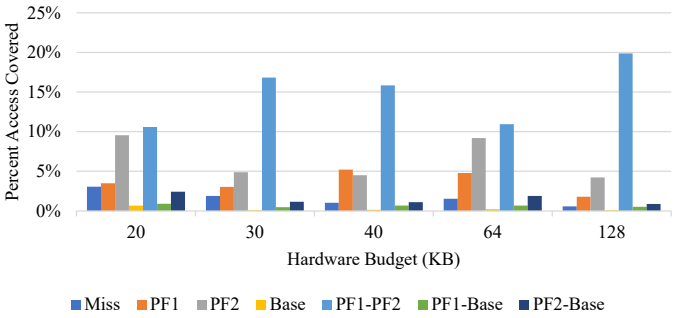

Fig. 8: Coverage breakdown for Composite-2 prefetchers.

2) Measuring Subprefetchers' Individual Contributions:
Miss: A miss occurs for both prefetchers and in the baseline cache. This is generally a small percentage of accesses (1-2%) for any size of composite-2, indicating that the prefetchers or the baseline cache without prefetching cover most accesses.

PF1 Hit (PF1): This is the number of misses covered by unique prefetches from subprefetcher-1.

PF2 Hit (PF2): This represents the number of misses covered by subprefetcher-2.

Baseline Hit (Base): A hit occurs only for the baseline, without prefetching, indicating that the subprefetchers' behaviors cause a harmful eviction resulting in a cache miss. This scenario occurs $\leq$1% for all hardware budgets.

PF1 and PF2 Hit (PF1-PF2): A hit occurs for both subprefetchers, showing an overlap between the prefetches selected by the subprefetchers.

PF1 and Baseline Hit (PF1-Base): A hit occurs for both the baseline and subprefetcher-1, meaning that subprefetcher-2 caused a harmful eviction otherwise covered in the baseline.

PF2 and Baseline Hit (PF2-Base): A hit occurs for subprefetcher-2 and the baseline, indicating subprefetcher-1 caused a harmful eviction.

PF1, PF2 and Baseline Hit: A hit occurs for all three indicating the prefetchers are retaining useful cache lines that hit in the baseline system. This is the most common occurrence for more than 70% of accesses. These are not included in Figure 8 to improve the visibility of the other scenarios.

We find that the highest occurring scenario is a hit for the baseline and both subprefetchers. This is expected as the prefetchers generally try to cover misses with high accuracy to avoid thrashing the L1-I cache. This is supported by the low number of misses between the subprefetchers and baseline and the low number of unique baseline hits, indicating that the prefetchers avoid harmful evictions. The number of unique hits for each subprefetcher varies at different sizes because

each size contains a different subset of prefetchers. Interestingly, subprefetchers that individually perform better than their partnered subprefetcher (i.e., FNL+MMA vs. D-JOLT) have a higher number of unique hits than the other subprefetcher. As a subprefetcher's size increases to 64KB (128KB of total state), the number of unique hits is lower since each subprefetcher has enough storage to capture an application's behavior resulting in a higher number of overlapped hits between subprefetchers.

V. Conclusion

Instruction prefetchers' performances are limited by hardware overhead constraints but do not gain increased performance with larger hardware budgets. Composite prefetching allows for higher performance at lower hardware budgets by combining the coverage of different complex prefetchers but is challenging to design effectively without making components targeting specific behaviors. We demonstrate a framework for selecting and integrating state-of-the-art complex prefetchers to find the best performing combination at various hardware budgets in a "plug-and-play" fashion that lightens the burden of tailor-making components for specific behaviors.

Our framework provides the basis for future work designing composite hardware prefetchers using heterogeneously sized components. A potential optimization is to share metadata storage between prefetchers and dynamically allocate storage to prefetchers that excel in predicting specific program phases.

Future work in composite prefetching may also explore selecting subprefetchers on the fly based on an application's specific characteristics. Software analysis of an application can provide hints to the hardware on the most appropriate subprefetchers for specific hardware, such as accelerators.

VI. Acknowledgements

This work was supported in part by Semiconductor Research Corporation (SRC), NSF grants CNS-193806 and CCF-1912617, European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 819134), Ramón y Cajal Research Contract (RYC2018-025200-I), and generous gifts from Intel Corporation.

References

[1] R. Kumar, B. Grot, and V. Nagarajan, "Blasting through the front-end bottleneck with shotgun," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: ACM, 2018, pp. 30–42. [Online]. Available: http://doi.acm.org/10.1145/3173162.3173178

[2] S. Kanev, J. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a Warehouse-scale Computer," in *ISCA '15 Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2014, pp. 158–169.

[3] S. Mirbagher-Ajorpaz, E. Garza, S. Jindal, and D. A. Jiménez, "Exploring predictive replacement policies for instruction cache and branch target buffer," in *45th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2018, Los Angeles, CA, USA, June 1-6, 2018*, 2018, pp. 519–532. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00050

[4] A. J. Smith, "Sequential program prefetching in memory hierarchies," *Computer*, vol. 11, no. 12, p. 7–21, December 1978. [Online]. Available: https://doi.org/10.1109/C-M.1978.218016

[5] B. Falsafi and T. F. Wenisch, *A Primer on Hardware Prefetching*, 2014.

[6] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo, "The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling," *IBM Journal of Research and Development*, vol. 11, no. 1, pp. 8–24, 1967.

[7] J. Pierce and T. Mudge, "Wrong-path instruction prefetching," in *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 29. USA: IEEE Computer Society, 1996, p. 165–175.

[8] S. Srinath, O. Mutlu, H. Kim, and Y. 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, pp. 63–74.

[9] S. Kondguli and M. Huang, "Division of labor: A more effective approach to prefetching," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 83–95.

[10] A. Ros and A. Jimborean, "The entangling instruction prefetcher," https://research.ece.ncsu.edu/ipc/wp-content/uploads/2020/05/eip_final.pdf.

[11] T. Nakamura, T. Koizumi, Y. Degawa, H. Irie, S. Sakai, and R. Shioya, "D-Jolt: Distant Jolt Prefetcher," https://research.ece.ncsu.edu/ipc/wp-content/uploads/2020/05/D-JOLT.pdf.

[12] D. A. Jiménez, G. Chacon, N. Gober, and P. V. Gratz, "Branch agnostic region searching algorithm," https://research.ece.ncsu.edu/ipc/wp-content/uploads/2020/05/bar\%C3\%A7a.pdf.

[13] A. Seznec, "The fnl+mma instruction cache prefetcher," https://research.ece.ncsu.edu/ipc/wp-content/uploads/2020/05/FNLMMA-final.pdf.

[14] N. Gober, G. Chacon, D. A. Jiménez, and P. Gratz, "The temporal ancestry prefetcher," https://research.ece.ncsu.edu/ipc/wp-content/uploads/2020/05/tap_final.pdf.

[15] A. Ansari, F. Golshan, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Mana: Microarchitecting an instruction prefetcher," https://research.ece.ncsu.edu/ipc/wp-content/uploads/2020/05/mana.pdf.

[16] V. Gupta, N. S. Kalani, and B. Panda, "Run-jump-run: Bouquet of instruction pointer jumpers for high performance instruction prefetching," https://research.ece.ncsu.edu/ipc/wp-content/uploads/2020/05/JIP.pdf.

[17] P. Michaud, "Pips: Prefetching instructions with probabilistic scouts," https://research.ece.ncsu.edu/ipc/wp-content/uploads/2020/05/pips_final.pdf.

[18] A. Ansari, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Divide and conquer frontend bottleneck," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 65–78.

[19] S. Pakalapati and B. 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, pp. 118–131.

[20] M. Bakhshalipour, M. Shakerinava, F. Golshan, A. Ansari, P. Lotfi-Karman, and H. Sarbazi-Azad, "A survey on recent hardware data prefetching approaches with an emphasis on servers," 2020.

[21] A. Ros and A. Jimborean, "A cost-effective entangling prefetcher for instructions," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 99–111.

[22] "Champsim simulator," http://github.com/ChampSim/ChampSim.

[23] G. Reinman, B. Calder, and T. Austin, "Fetch Directed Instruction Prefetching," in *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, 1999, pp. 16–27.

[24] Y. Ishii, J. Lee, K. Nathella, and D. Sunwoo, "Re-establishing fetch-directed instruction prefetching: An industry perspective," in *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2021, pp. 172–182.

[25] *The 1st Championship Value Prediction Competition (CVP-1), http://www.microarch.org/cvp1*. International Symposium on Computer Architecture, June 2018.

[26] J. Kim, S. H. Pugsley, P. V. Gratz, A. N. Reddy, C. Wilkerson, and Z. Chishti, "Path Confidence-based Lookahead Prefetching," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12.