Hierarchical Prefetching: A Software-Hardware Instruction Prefetcher for Server Applications

Tingji Zhang
Tsinghua University
Beijing, China
ztj22@mails.tsinghua.edu.cn

Boris Grot
University of Edinburgh
Edinburgh, United Kingdom
Huawei Research
Zurich, Switzerland
boris.grot@ed.ac.uk

Wenjian He
Huawei Technologies Co., Ltd.
Shanghai, China
wheac@connect.ust.hk

Yashuai Lv
Huawei Technologies Co., Ltd.
Beijing, China
lvyashuai1@huawei.com

Peng Qu
Tsinghua University
Beijing, China
qp2018@mail.tsinghua.edu.cn

Fang Su
Huawei Technologies Co., Ltd.
Shenzhen, China
fang.su@huawei.com

Wenxin Wang
Tsinghua University
Beijing, China
wenxin-w23@mails.tsinghua.edu.cn

Guowei Zhang
Huawei Technologies Co., Ltd.
Shanghai, China
zhangguowei9@hisilicon.com

Xuefeng Zhang
Tsinghua University
Beijing, China
zhang-xf21@mails.tsinghua.edu.cn

Youhui Zhang
Tsinghua University
Beijing, China
Zhongguancun National Laboratory
Beijing, China
zyh02@tsinghua.edu.cn

Abstract

The large working set of instructions in server-side applications causes a significant bottleneck in the front-end, even for high-performance processors equipped with fetch-directed instruction prefetching (FDIP). Prefetchers specifically designed for server scenarios typically rely on a record-and-replay mechanism that exploits the repetitiveness of instruction sequences. However, the efficacy of these techniques is compromised by discrepancies between actual and predicted control flows, resulting in loss of coverage and timeliness.

This paper proposes Hierarchical Prefetching, a novel approach that tackles the limitations of existing prefetchers. It identifies common coarse-grained functionality blocks (called Bundles) within the server code and prefetches them as a whole. Bundles are significantly larger than typical prefetch targets, encompassing tens to hundreds of kilobytes of code. The approach combines simple software analysis of code for bundle formation and light-weight hardware for record-and-replay prefetching. The prefetcher requires under 2KB of on-chip storage by keeping most of the metadata in main memory. Experiments with 11 popular server workloads reveal that Hierarchical Prefetching significantly improves miss coverage and timeliness over prior techniques, achieving a 6.6% average performance gain over FDIP.

CCS Concepts: • **Computer systems organization → Architectures**.

Keywords: Microarchitecture, Front-End Bottleneck, Instruction Prefetching

ACM Reference Format:
Tingji Zhang, Boris Grot, Wenjian He, Yashuai Lv, Peng Qu, Fang Su, Wenxin Wang, Guowei Zhang, Xuefeng Zhang, and Youhui Zhang. 2025. Hierarchical Prefetching: A Software-Hardware Instruction Prefetcher for Server Applications. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3676641.3716260

ASPLOS '25, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1079-7/2025/03
https://doi.org/10.1145/3676641.3716260

1 Introduction

The front-end bottleneck is a long-standing problem for server processors stemming from complex software stacks in server workloads that encompass intricate application logic, extensive use of libraries, language runtimes, and kernel modules. The result is instruction working sets that far exceed the capacity of the instruction cache, often even the L2 cache [36, 42, 43, 57]. I-Cache misses caused by the large

instruction working set result in significant front-end stalls that compromise performance. Moreover, studies [17, 34, 38] indicate that the growth rate of instruction working sets in commercial server deployments are growing fast, leading to a continuous exacerbation of front-end issues.

Instruction prefetching is widely used to address front-end issues in server processors. Fetch directed instruction prefetching (FDIP) leverages existing branch prediction mechanisms to anticipate future instruction accesses [20, 25, 31, 36, 42, 43, 49], and has been widely adopted across mainstream server chips [12, 20, 26, 31, 48]. But previous studies have shown that FDIP's effectiveness is significantly hindered by BTB misses, which prevent the branch predictor from discovering control flow discontinuities [25, 37, 43, 57].

Researchers have proposed techniques specifically aimed at the front-end bottleneck in servers. These tend to rely on some form of a record-and-replay mechanism and work by identifying repeated instruction sequences or function calls for recording and subsequent prefetching (aka replaying) [14, 19, 21, 23, 35, 36, 40, 56]. The trigger for replaying a particular sequence of instructions is typically an instruction address or a signature, such as a hash of several recent function call addresses. For the sake of timeliness, some approaches, such as EIP [50], attempt to identify the right trigger for a subsequent miss, taking into account the latency of each prefetch. These triggers effectively capture the local contextual information, which may be sufficient to prefetch a small number of upcoming cache blocks provided that the control flow stays on the anticipated path. However, each trigger may cause associated addresses from all historical executions to be prefetched, causing inaccurate prefetches if those prefetches are no longer useful. To avoid such inaccuracies, the prefetcher must detect changes in the execution phase and only issue prefetches that align with the current functional context.

Context switching can also prevent the prefetcher from improving coverage and timeliness by simply increasing its depth to deliver more likely-useful cache blocks. Our tests indicate that the accuracy of these prefetchers [14, 21, 50] rapidly declines as the prefetch depth increases, limiting their ability to achieve further miss coverage. Thus, the short lookahead of existing instruction prefetchers for servers is a fundamental impediment to their efficacy.) Additionally, these prefetchers have to store a significant amount of metadata on-chip to ensure both timeliness and coverage at lower prefetch depth. Despite efforts to reduce the on-chip storage overhead [14, 21, 23, 24, 35, 36, 40], the state-of-the-art prefetchers [14, 21, 50] still requires 15KB-40KB of storage per core.

The central premise of this work is that effective front-end prefetching for server workloads needs to happen at a coarser granularity than the one exploited by existing instruction prefetchers in order to maintain good coverage and timeliness despite unpredictable variations in control flow. Toward that end, we make a critical insight that server applications exhibit high commonality at a global level that can be exploited for prefetching.

Specifically, we find that server applications commonly execute sequences of high-level *functionalities*, each of which may span multiple functions at the code level. While the sequence of functionalities for a given request type and/or input may be difficult to predict, the individual functionalities exhibit stable instruction working sets spanning 10s-100s of KB of code. Each of these functionalities presents an excellent target for coarse-grained prefetching, which is precisely the opportunity that this paper identifies and exploits.

To capitalize on the observation above, this work proposes Hierarchical Prefetching, a software/hardware cooperative scheme to achieve high prefetch coverage, accuracy and timeliness without profiling or programmer's involvement. Hierarchical Prefetching identifies the code sequences forming high-level functionalities (referred to as Bundles) and passes Bundle starting points to the hardware, which then records the actual instruction working set of each Bundle at runtime. Subsequently, when an instruction identified as an entry point to a Bundle is encountered, the recorded footprint of that Bundle is streamed into the I-Cache.

Bundles are large, over 10KB in size for the studied applications, and are prefetched non-speculatively; i.e., prefetching starts only when an instruction tagged as an entry point to the Bundle is committed. Crucially, prefetching is unaffected by local control flow variation within a Bundle. If a cache block fetched by the core is not in the Bundle's recorded footprint, the prefetcher does not take any corrective actions (though the recorded footprint is updated for next time). This ensures that the prefetcher can run far ahead of the core within each Bundle's execution and can deliver the vast majority of needed cache blocks on time despite potential variations in intra-Bundle control flow across executions. Hierarchical Prefetching only emits prefetches for cache blocks that were encountered in the most recent execution of the same Bundle, ensuring that the prefetched content aligns with the currently executing functionality, thereby improving accuracy. Replaying only the blocks observed in the most recent execution of the bundle also helps quickly unlearn execution paths that only occur sporadically, further improving accuracy.

The Bundle identification algorithm uses the static call graph of the application to delineate the Bundles and identify each Bundle's entry point. The algorithm runs as part of the linking process, which allows it to incorporate dynamically linked libraries into the analysis. Our results indicate that the algorithm is effective at forming Bundles, and there is a high degree of similarity (over 80% at the cache block level) between the successive reuses of a Bundle.

Our design of the Hierarchical Prefetcher stores the instruction footprints of Bundles in main memory, requiring merely 1.94KB of on-chip storage per core for recording and replaying. A comprehensive evaluation shows that the scheme is highly effective at eliminating instruction cache misses, demonstrating better coverage and timeliness compared to prior techniques. Because of that, Hierarchical Prefetching improves the performance of a range of server applications by 6.6% on average over FDIP.

To summarize, we make the following contributions:

- We show that existing prefetchers are compromised by control flow variations between the recorded history and runtime execution.
- We observe that server workloads feature coarse-grained functionalities, which demonstrate high commonality across executions and encompass 10s to 100s of KB of code.
- We introduce Hierarchical Prefetcher, which statically identifies these coarse-grained functionalities in the code, forming Bundles. At runtime, it performs bulk record-and-replay prefetching at the Bundle level.
- Experiments on 11 popular server applications show that with just 1.94KB of on-chip storage, Hierarchical Prefetching yields a 6.6% performance increase over FDIP by improving prefetch coverage and timeliness against state-of-the-art instruction prefetchers that achieve at most 4% improvement while requiring tens of KB of on-chip storage.

2 Background

2.1 BTB-directed Prefetching

FDIP [20, 25, 30, 31, 36, 42, 43, 49] leverages existing branch prediction mechanisms to predict future accesses and perform prefetching, eliminating the need for substantial additional metadata overhead. FDIP utilizes a decoupled front-end architecture where the future instruction addresses predicted by the branch predictor are pushed into the Fetch Target Queue (FTQ), which is then used to issue prefetch requests.

A key limitation of FDIP is its reliance on the BTB for discovering upcoming branches, which can become compromised due to limited BTB capacity. Another source of potential performance loss is the accuracy of the branch predictor. Despite these limitations, FDIP has become the standard front-end design for high-performance server CPUs [12, 20, 26, 31, 48] because of its low overhead. We use FDIP serves as the baseline for all experiments of this paper.

2.2 Temporal Streaming Prefetching

The temporal prefetcher has been a predominant prefetching technology to mitigate the front-end bottleneck in server architectures over the past decade. Temporal prefetchers proactively eliminate future instruction cache misses by recording historical sequences of accesses or misses and replaying them. TIFS [24] first applied this concept by recording sequences of misses. PIF [23] then made further improvements by recording sequences of instruction accesses, demonstrating that the record and replay mechanism can eliminate most misses. However, due to the need to record sufficient historical sequences on chip, PIF introduced significant area overhead, with a metadata budget reaching 200 KB per core.

Subsequent research [14, 35, 36] has focused on reducing metadata overhead through various means. The state-of-the-art MANA reduces the required on-chip metadata storage to 15 KB by using cache structures to record history and compressing trigger addresses.

Temporal prefetchers operate in a fine-grained manner. Each prefetch stream (a set of instruction cache blocks) is linked to a trigger address; when the trigger address is encountered at runtime, prefetching for the linked stream is initiated. Our testing shows that the majority of the streams are short, rarely exceeding 10 cache lines. Prior work has also identified this limitation of temporal stream prefetchers [51].

2.3 Caller-callee Prefetching

Some prior works have suggested performing record and replay prefetching at the granularity of function calls [13, 21, 40, 46] as a way of improving prefetch accuracy and reducing metadata storage compared to temporal streaming.

CGP, one of the earliest works in this area, effectively eliminates immediate post-function call I-Cache misses by prefetching one function ahead each time. RDIP [40], a subsequent study, leverages recent function call information to offer performance comparable to PIF. Observing that the Return Address Stack (RAS) can reflect the current state of the program to a certain extent, RDIP utilizes the hashes of the top four entries of the RAS to generate a signature. Throughout its execution, RDIP records misses that follow these signatures and prefetches when they recur. This method is metadata-intensive, requiring 60KB of storage per core.

EFetch [21] further enhances caller-callee prefetching for event-driven applications by employing a more precise signature that combines three recent function calls with an Event-ID. Once a new signature is generated, the next callee function will be predicted and prefetched. EFetch achieves performance surpassing both CGP and RDIP and requires under 40KB of on-chip metadata.

Compared to temporal prefetchers, caller-callee prefetchers record/replay at a coarser granularity of entire functions. This helps caller-callee prefetchers achieve similar or better performance with less metadata storage versus temporal prefetchers. However, given that they prefetch only one callee at a time, and because the size of a typical function is small, the lookahead of caller-callee prefetchers is limited, which adversely affects their performance.

2.4 Correlating Prefetching

EIP [50] is a state-of-the-art instruction prefetcher and winner of the IPC-1 instruction prefetching competition [3]. It is

a correlating prefetcher that emphasizes prefetch timeliness by entangling basic blocks that missed in the L1-I with an instruction executed prior to the observed miss latency. A cost-performance optimized EIP configuration uses a 4K-entry entangle table, which requires 40KB of storage per core [50].

2.5 Scenario-Based Prefetching

Some other studies are application-specific or scenario-specific. STREX [16] and SLICC [15] leverage the repetitiveness of OLTP transactions for instruction reuse. PTask [33], Jukebox [51] and Ignite [52] record and replay for applications with frequent system calls or serverless scenarios. I-SPY [38], Ripple [39], Twig [37] and Thermometer [57] require offline profiling.

3 Motivation

3.1 Understanding Application Characteristics

We explore opportunities to enhance prefetchers by studying server application characteristics. Taking TiDB (PingCAP's open-source distributed SQL database [8]) as an example, Figure 1 depicts its processing stages in the life cycle of a typical SQL statement during TPC-C execution. For each stage, the figure shows the average instruction working set computed based on the number of accessed instruction cache blocks during the execution of the stage.

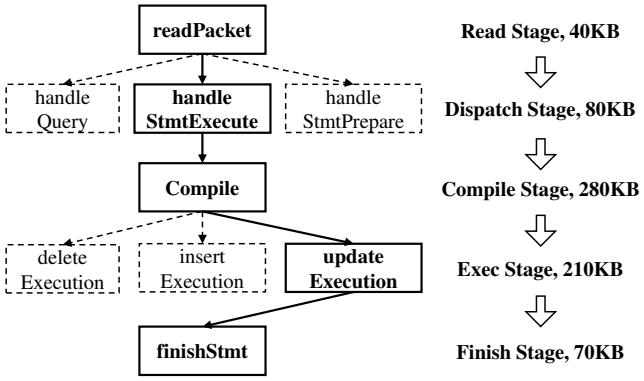


Figure 1. The key stages of a statement in TiDB and the average footprint during the execution of TPC-C for each stage. Dashed arrows indicate potential functional flows for other types of statements.

Request processing in TiDB typically progresses through several stages including *Read*, *Dispatch*, *Compile*, *Exec* and *Finish*. Each of these stages has a sizable instruction working set, ranging from 40KB to 280KB. Most statements go through all of the stages, resulting in a large instruction working set per statement.

Furthermore, a single stage may encompass various functional routines (i.e., code sequences) for different types of statements. Take the *Exec* stage as an example. It selects the appropriate functionality, based on input data and the type of statement, from among several routines such as *deleteExecution*, *insertExecution*, or *updateExecution*. Because these routines are executed only during a specific stage and a specific type of statement, and the total instruction working set is huge, by the time one of these routines is executed again, numerous other routines will have already executed since the last time the routine ran. Taking *updateExecution* as an example, even during the periods with intensive update operations while running TPC-C, the average interval between two successive invocations of *updateExecution* is 1.23 million cycles / 1.51MB instruction footprint. Not surprisingly, it is impossible to maintain the instruction working set in the I-Cache and even in the L2.

3.2 Limitations of Existing Instruction Prefetchers

Intuitively, discovering and tracking the repeated execution sequences at a coarse granularity (i.e., in order to achieve a large *prefetch distance*) requires a relatively global application-level horizon. A large prefetch distance, defined as the number of cache lines between the prefetch target and the trigger, can improve timeliness and coverage. In practice, being able to prefetch over large distances is challenging because the larger the distance, the more likely the actual control flow to diverge from the expected control flow, resulting in diminished accuracy and coverage. For this reason, existing prefetch techniques struggle to obtain high accuracy prefetch at large prefetch distances.

To validate this intuition, we tested the performance of state-of-the-art prefetchers under different prefetch distances by adjusting their look-ahead settings, with larger look-ahead values leading to greater distances. Since EIP does not have a look-ahead parameter, we grouped and analyzed its prefetches based on distance ranges. Accuracy is computed as the percentage of prefetches that yield an L1-I hit for a demand fetch, while coverage is the percentage of demand misses eliminated due to prefetches. Note that we calculate accuracy and coverage on top of the FDIP baseline, meaning only the misses remaining after FDIP prefetching are considered.

As shown in Figure 2, all prefetchers exhibit a decline in accuracy as the distance increases. Additionally, as shown in Figures 2a and 2b, once the look-ahead exceeds 4 spatial regions for MANA and 7 functions for EFetch, their coverage fails to improve, leading to a rebound in I-Cache misses. For EIP, the prefetch distance is determined by miss latency and cannot be adjusted; therefore, we grouped EIP's prefetches by distance and present only its accuracy as a function of the distance.

We further compare the overall accuracy and coverage of the three SOTA prefetchers in Figure 3. The X-axis in the figure shows the average prefetch distance, in cache blocks, observed for each of the prefetchers. The three prefetchers demonstrate overall accuracy ranging from 30% to 58%,

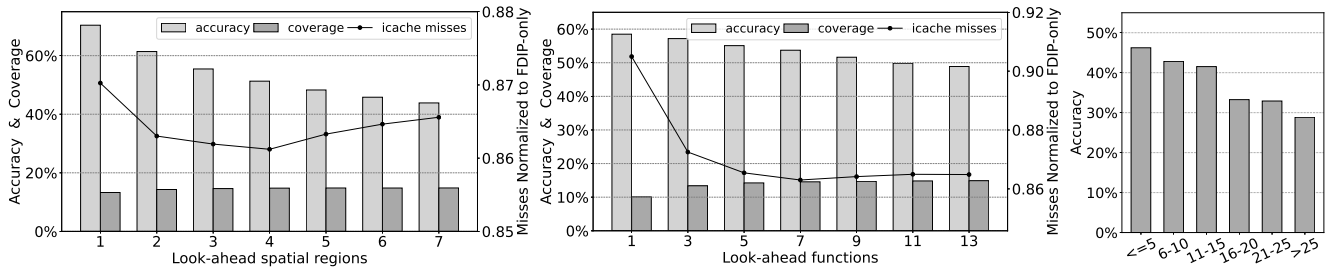


Figure 2. Analysis of fine-grained prefetchers. (a) Impact of MANA's look-ahead spatial regions. (b) Impact of EFetch's look-ahead function calls. (c) EIP's accuracy under different prefetch distances (cache blocks).

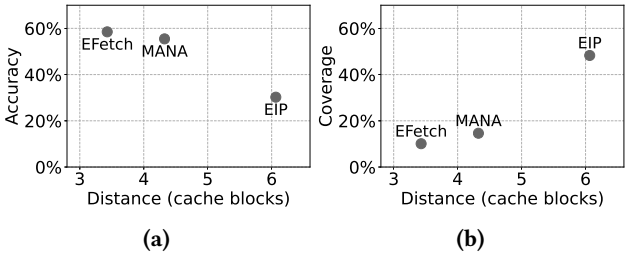


Figure 3. (a) Accuracy and (b) coverage for fine-grained instruction prefetchers as a function of prefetch distance.

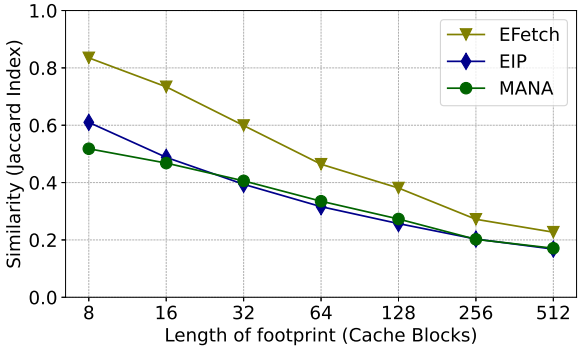


Figure 4. Average similarity (Jaccard index) between footprints of varying lengths following adjacent occurrences of the same trigger/signature across all our applications.

with accuracy inversely correlated to average prefetch distance. EFetch achieves the highest accuracy with the lowest prefetch distance. The trend is opposite for coverage, where we observe that the larger the average prefetch distance, the better the coverage. This confirms the hypothesis that a larger prefetch distance is a must for achieving high instruction miss coverage.

Notably, we observe that both MANA and EFetch exhibit relatively low miss coverage, with the better-performing one (MANA) eliminating less than 20% of the instruction misses over FDIP. The reason for such limited gains from existing prefetchers is that the FDIP baseline is able to cover largely the same misses, which are the shorter-distance ones, and all of these prefetchers (including FDIP) are susceptible to control flow variations making them less effective at covering longer-range misses as discussed above. While EIP achieves higher coverage, its low accuracy causes excessive incorrect prefetches, thereby polluting the cache and limiting miss reduction. Our findings corroborate recent work, that also observed limited benefit from SOTA prefetchers in the presence of FDIP. Nonetheless, we note that the combination with FDIP is more effective than any of the prefetchers stand-alone in both coverage and performance.

One part of the challenge in predicting upcoming control flow for prefetching purposes is that some branches are just fundamentally difficult to predict. However, in many cases, the set of upcoming instruction blocks can be predicted given a sufficient understanding of the program context. For instance, the set of instruction cache blocks touched in a deeply nested function may heavily depend on the call path leading to the execution of that function. If the prefetcher is aware of the global context, it may be able to do a better job anticipating the set of instruction cache blocks for prefetching.

Existing record-and-replay prefetchers trigger prefetching using the address of an instruction or a function call or a hash of a few recent function calls. The underlying assumption is that the instruction sequence following a trigger will resemble the previously observed sequence after the same trigger. Alas, that assumption is often incorrect due to the weak contextual representation of the triggers.

We studied the average similarity between sets of cache blocks (footprints) following occurrences of the same trigger in EFetch, MANA and EIP across a set of server applications described in Section 6.2. To assess the similarity, we used Jaccard index[1] while varying the size of the footprint from 16 to 512 cache blocks.

Results of the study are shown in Figure 4. MANA and EIP exhibit rapidly diminishing similarity as the size of the footprint increases. Notably, EFetch achieves higher similarity than MANA/EIP due to its use of a more contextually representative signatures. However, all three prefetchers have a similarity below 0.5 at 64 cache blocks, indicating that a

[1]The Jaccard index is used to compare the similarity between two samples, A and B, as the intersection of A and B divided by the union of A and B. [32]

deeper prefetch will necessarily result in both lower accuracy and lower coverage.

Summary: Existing instruction prefetchers have a fundamental deficiency: they either cannot cover the long miss latency with shallow prefetching or lose accuracy by prefetching further ahead. To fundamentally break through this dilemma, it is necessary to enhance the prefetcher's ability to predict future instruction sequences over a longer distance. To do so, prefetchers should encompass more global information to expose and exploit the longer-range reuse inherent in applications.

4 Hierarchal Prefetching Overview

Opportunities for better prefetching lie in understanding and exploiting the repetition of coarse-grained regions of code in server applications across long reuse distances, which is a challenge for current fine-grained prefetchers. Server applications typically follow a request-response pattern, where each request undergoes a similar processing flow. A straightforward idea is to perform record-and-replay on a per-request basis, recording the instruction stream during the request's execution and prefetching it before the next.

However, this approach often fails in practice: different requests may lead to different specific functionalities due to varying request types and input data, resulting in divergent instruction sequences. As shown in Figure 1, despite running in the same processing framework, different requests may diverge in the *Dispatch* and *Exec* phases towards different functionalities. These relatively few but significant (coarse-grained) divergence points of control flow prevent sufficient similarity between multiple executions of a request to support accurate prefetching. Furthermore, the MB-level instruction working set poses a challenge to prefetching the entire request-handler at once as its code footprint will overwhelm the I-Cache capacity.

We observe that the processing procedure for a given request type can be viewed as a combination of specific subsets of functionalities, such as reading a packet, compiling a statement, or updating a database index. Indeed, each functionality typically encompasses a precise and definitive set of tasks, leading to a highly stable instruction footprint. Moreover, these functions are executed each time a particular type of request or a given input is encountered. By identifying common code regions as well as the points of divergence between these functionalities, we can gain insights into the program's behavior at a coarse granularity. This understanding can then be leveraged to drive prefetching.

The crux of our approach is to partition the entire call graph of an application into a series of coarse-grained consecutive functionalities, per above. By doing so, we can leverage the stability of these functionalities to perform coarse-grain history-driven prefetching.

What about the divergence points in an application's call graph? If a call graph node has multiple child nodes with large footprints, it suggests that vastly different instruction sequences could be initiated from this node, leading to significant variations in the instruction footprints that follows. Each path stemming from such a divergence point signifies the start of a distinct stable instruction sequence that persists until the next point of divergence.

Given this understanding, we can pinpoint all divergence points that might significantly influence the subsequent instruction footprint, thus partitioning the entire call graph. We define the stable acyclic graph of functions between the major divergence points as *Bundles*. Note that minor divergence points, such as if-then-else construct (that might include function calls) with a small instruction footprint, are incorporated into their constituent Bundles.

Figure 5 illustrates the partitioning and execution process with an example. In the figure, each node represents a function, and the number beneath each function label is its *reachable size*, defined as the total instruction working set for that function *and* all functions reachable from it (e.g., the reachable size of function A is the sum of the instruction working sets of functions A, B, C, D, and E taken together). This size is used to determine whether a given divergence is significant (in which case it starts a new Bundle) or not.

As Figure 5a shows, the call graph is analyzed from the starting node, A. At some point, a divergence point is identified, with the two paths (B and C) each having a reachable size that exceeds the threshold for Bundle formation (200KB in this example). Subsequently, the *call* and *return* instructions for B and C are tagged as potential entry points for Bundles. Note that although D meets the threshold for Bundle formation, it is not considered to be a divergence point due to the small difference in reachable size with its parent node C. Also note that because the root node, A, exceeded the threshold, it is also tagged. During execution, the occurrence of a tagged call or return triggers the start of a new Bundle, which continues until a subsequent Bundle begins. As noted above, a Bundle may incorporate multiple functions from the application's call graph; for example, Bundle3 depicted in Figure 5b includes the execution of functions C, D, and E.

By recording instruction history at the granularity of Bundles, we can use it for prefetching at a coarse grain thus achieving high coverage and good timeliness. This gives rise to the notion of Hierarchical Prefetching – at the top level, a series of Bundles represents the coarse-grained execution flow of the application, while at the lower level, the individual instruction cache blocks within each Bundle are tracked and prefetched in bulk.

5 System Design

We introduce *Hierarchical Prefetching*, a hardware-software collaborative prefetching scheme aimed at uncovering and

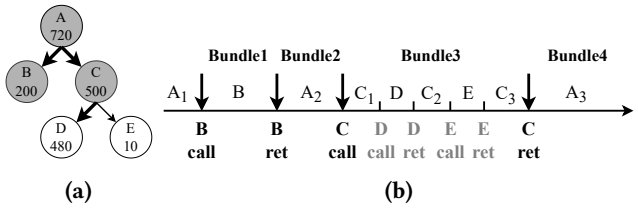


Figure 5. (a) Example call graph, with numbers indicating a node's reachable size (KB). Grey circles indicate entry points of Bundles (divergence threshold set at 200KB). (b) Function execution flow.

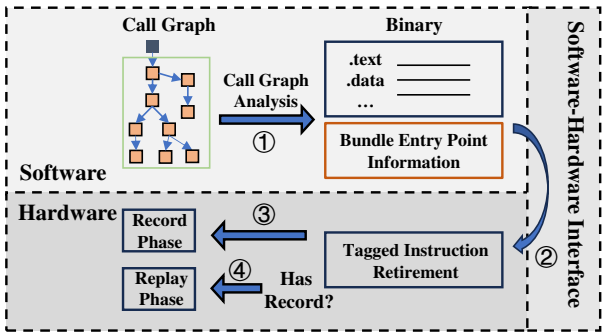


Figure 6. Overview of Hierarchical Prefetching.

leveraging repetitive instruction sequences with coarse granularity (aka Bundles) within server applications. Figure 6 overviews the design.

First, the proposed algorithm (Section 5.1) identifies entry points of Bundles by analyzing the software's call-graph ①, then stores the related information in the program's binary. Subsequently, the hardware-software interface tool (Section 5.2), e.g. the linker or the loader, utilizes this information to locate the function call and return instructions that represent the Bundle entry points and tags these instructions for hardware identification ②.

During program execution, the hardware prefetcher (Section 5.3) detects the tagged instructions during the commit stage. Whenever it encounters a tagged instruction, which indicates a Bundle entry point, it calculates the Bundle ID and starts the *Record* process ③ (Section 5.3.4) to store the ID and addresses of executed instructions of this Bundle, until another tagged instruction is encountered or the record length exceeds a predetermined threshold. If a matching historical record is located, the prefetcher also activates the *Replay* process ④ (Section 5.3.5) to initiate prefetching of instruction cache blocks into the L1-I cache.

5.1 Software Algorithm

The software process is illustrated by Algorithm 1. It consists of three steps: constructing the call-graph, calculating reachable size, and identifying Bundles' entry points.

Call-graph construction: We first traverse all functions to retrieve their names, code sizes, and address ranges. By identifying all call instructions within these functions, we establish parent-child relationships to construct the call graph[2].

Algorithm 1 Partition Bundles.

```
1:  procedure GetBundleEntries(binary, threshold)
2:      call_graph = BuildCallGraph(binary)
3:      reachable_size = GetReachableSize(call_graph)
4:      for func, size in reachable_size do
5:          if size < threshold then
6:              continue
7:          end if
8:          if any(func.fathers.size - size > threshold) then
9:              Entries.add(func)
10:         end if
11:     end for
12:     return Entries
13: end procedure
```

Calculating Reachable Size: We initiate searches from each function and compute their reachable sizes. This can be accomplished using either a depth-first or a breadth-first search algorithm.

Identifying Bundles' Entry Points: For each node in the call-graph, if it has multiple child nodes with reachable sizes exceeding the threshold, *n* KB, consider these child nodes as potential entry points for Bundles. In practice, we slightly relax this requirement, only requiring a child node's size to exceed the threshold and the difference in size between the node and this child to be greater than the threshold. Specifically, the root nodes are regarded as a Bundle as long as they meet the size requirement. In practice, we set 200KB as the divergence threshold to ensure that these entry points appear at an appropriate frequency during execution.

While 200KB per Bundle may appear as a particularly large amount of code, especially considering the typical size of an I-Cache, we note that this is the *static* size. Prefetching within each Bundle happens based on *dynamic* footprints recorded during a Bundle's execution, which tend to be 3-10 times smaller than the static footprint.

5.2 Software-Hardware Interface

The software-hardware interface is tasked with transmitting Bundle information, identified by the software algorithm, to the hardware.

From the hardware's perspective, each Bundle represents a sequence of instructions including branches, and multiple Bundles executed in succession constitute the entire execution flow. Thus, the interface only needs to explicitly tag the entry points of all possible Bundles. In ISAs like X86(-64) [27] or AArch64 [53], where call/return instruction formats reserve at least 2 unused bits, we avoid inserting new

[2]Static call graphs tend to overestimate the actual graphs. Nevertheless, they have worked well in our design.

instructions by simply using one of the reserved bits identifying Bundle entry points. In ISAs in which such reserved bits are not available, a new instruction would need to be added to identify a Bundle entry point. Since each Bundle spans 10s-100s of KB of code, the overhead of the additional instructions would have a negligible impact on both static and dynamic code size.

We construct the call-graph to identify Bundles during the linking process. A strength of this approach is that it naturally covers all dynamically linked code. Subsequently, we add a segment to the binary to record the addresses of the entry point instructions for the Bundles, akin to the .dynamic section in ELF files. We leverage the information encoded in this segment during the application loading phase to tag the call/return instructions corresponding to Bundle entry points by using the reserved bits as explained above. Note that our algorithm, performed during the linking and loading stages, is very fast, usually completing within seconds for a server application.

5.3 Hardware Support

At the hardware level, we record all retired instructions for each Bundle and assign them a Bundle ID hashed from the address of the next instruction following the tagged one. All Bundle records are stored in the main memory, in a dedicated memory space accessible by our prefetcher, referred to as the *Metadata Buffer* (② in Figure 7, see Section 5.3.2). The compressed historical records in the metadata buffer are allowed to be cached in the L3 cache (LLC), which the prefetcher accesses on a cache line basis through the existing LLC interface. To quickly determine the location of each Bundle's record, an on-chip *Metadata Address Table* (① in Figure 7, see Section 5.3.3) is used to associate the Bundle ID with the address of its head segment (elaborated below).

To minimize space overhead, we adopt a commonly-used compression technique [14, 23, 51] (Section 5.3.1) to store the instruction stream compactly as a sequence of *spatial regions* (④ in Figure 7); a spatial region contains a maximum of 32 contiguous cache blocks encoded as a base address and a bit vector. Further, to facilitate prefetching and memory management, the prefetcher divides the sequence of spatial regions into multiple segments (③ in Figure 7), each containing 32 spatial regions. Each Bundle can be viewed as a list of variable-length segments. Each segment serves as a prefetch unit, with the size chosen to ensure that each group of prefetch can fit within the L1-I cache capacity.

A key advantage of our approach is that we can store the prefetching metadata in main memory, thus avoiding most of the on-chip storage overhead associated with previous mainstream methods. This is feasible because the execution time of a Bundle typically extends to tens of thousands of cycles (Section 7.6), which allows us to tolerate the latency of accessing memory at the beginning of each Bundle. Because prefetching is decoupled from the execution (e.g., fine-grained variations between the prefetched instructions and the executed code do not affect the prefetcher), the prefetcher quickly gains and maintains a sufficient prefetch distance for latency hiding.

For multicore execution, we exploit the similarity in control flow (and hence, instruction block access sequences) among the different cores running a given server workload [35]. Thus, we share the metadata buffer across multiple cores and randomly select one core to generate the instruction history – an approach shown effective for reducing metadata volume without compromising performance [35].

5.3.1 Compression Buffer. Compression Buffer is a fully associative FIFO structure responsible for recording instruction streams using compact spatio-temporal encoding. Each entry in the Compression Buffer represents a spatial region, containing a base address and a bit vector where each bit corresponds to a cache block within that region. Whenever an instruction retires, the Compression Buffer first checks if the instruction's address falls within the range of an existing spatial region entry; if so, the bit corresponding to the cache block of the instruction within that entry is set. Otherwise, a new entry is created based on the address of the current instruction and pushed into the Compression Buffer, while the earliest entry is evicted and written to the *Metadata Buffer*.

Compression Buffer efficiently compresses address information by eliminating redundancy among recently executed instruction addresses. Additionally, the creation order of spatial regions is preserved, facilitating a replay that approximately mirrors the order of retired sequences. We use a 16-entry Compression Buffer, with each entry being a spatial region representing 32 contiguous cache blocks.

5.3.2 Metadata Buffer. The in-memory Metadata Buffer stores the spatial region sequences of all Bundles. It is divided into segments (each containing fix-sized spatial regions) and the basic structure is an implicit circular list. Each Bundle record is stored in one or multiple segments allocated from the buffer, organized into an implicit linked list. If the Metadata Buffer becomes full, segments will be sequentially reclaimed starting from the beginning, and any overwritten Bundles will be marked as invalid in the Metadata Address Table. We use a 512KB per-core in-memory Metadata Buffer.

5.3.3 Metadata Address Table. This table is an on-chip set-associative structure with LRU replacement designed to store the head segment address of each Bundle. Different Bundles may generate sequences of spatial regions of varying lengths, which are dynamically allocated during execution. Whenever a tagged instruction is encountered, the table is searched using the Bundle ID: if an entry is present, replay and record will initiate from the corresponding address in the Metadata Buffer; otherwise, new segment is allocated for

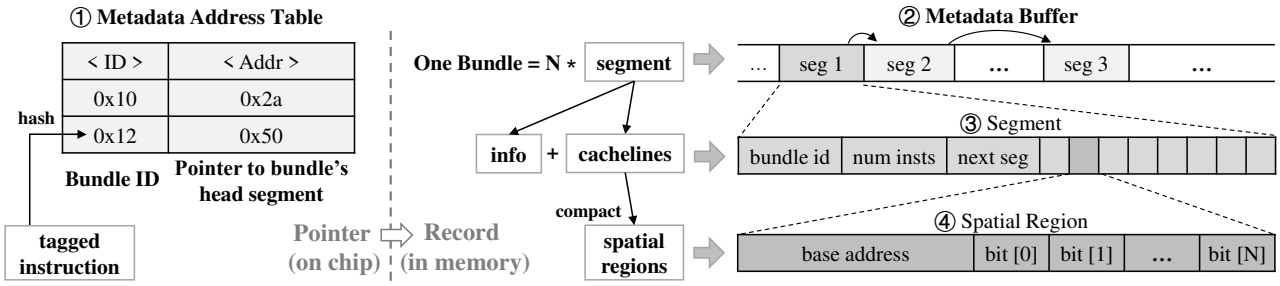


Figure 7. Bundle's logical structure and metadata organization.

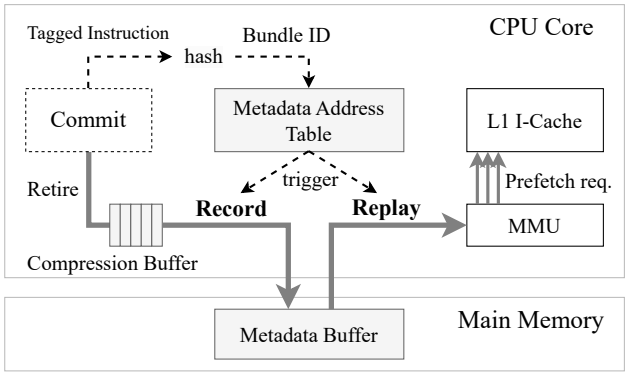


Figure 8. The record and replay process. Grey boxes indicates new structures and metadata related to our method.

the current Bundle ID, its address is recorded in the table, and the least recently used entry is evicted (if full).

Our default Metadata Address Table configuration has 512 entries and features 8-way set associativity. We use a 24-bit Bundle ID, and the pointer to metadata requires 11 bits to index a segment (0.36KB) in the 512KB in-memory Metadata Buffer. Each way requires an LRU bit, and each entry includes 18 bits for the tag, 11 bits for the pointer, and 1 bit for validity, resulting in 15872 bits (1.94KB) of on-chip storage.

5.3.4 Record. As shown in Figure 8, the record phase starts each time a tagged instruction is identified and ends when the next tagged instruction is encountered. If there is no valid record for the corresponding Bundle ID, new segments are allocated from the Metadata Buffer, and their spatial regions are written. If a record already exists, the current sequence is recorded in the existing segments, superseding the previous record. This is to avoid a pathology where the record is not representative for a given Bundle. If the new record's length exceeds that of the previous, additional segments are allocated as needed.

The record process allocates metadata in segments. The metadata for each segment contains a set of spatial regions as well as the following items of information (Figure 7):

- **next seg:** As multiple segments of a Bundle might not be contiguous, the pointer to the next segment is updated whenever a new segment is allocated. The implicit linked list supports this feature by default.
- **num insts:** The number of instructions executed from the start of the current Bundle is recorded when a new segment is created. This information is then used to pace the prefetching of the corresponding segment during replay.
- **Bundle ID:** We record the ID of each Bundle in its first segment. This value is used to maintain the integrity of the Metadata Address Table, allowing for the invalidation of the corresponding entry when rewrites occur in the Metadata Buffer.

The replay process will utilize these information to locate the next segment and manage the prefetch pace.

5.3.5 Replay. The replay phase (see Figure 8) begins when a tagged instruction is encountered and a matching Bundle ID is found in the Metadata Address Table. The prefetcher starts reading from the head segment of the Bundle whose address is recorded in the Metadata Address Table, progressively accessing the spatial regions within it. The spatial regions are sequentially loaded into a small FIFO, and their base addresses are dispatched to the TLB for address translation. Subsequently, the prefetch engine generates requests from lower to higher addresses, guided by the bit vector in the associated entry, and pushes these requests into the prefetch queue. As entries at the end of the FIFO are continually converted into prefetch requests and sent out, the prefetch engine continuously reads new cache lines of spatial regions, replenishing the FIFO.

Since the code footprint of a Bundle might exceed the capacity of the L1-I cache, we prefetch the instructions segment by segment to ensure that the prefetched content is timely and remains within the L1-I capacity limits. After processing all the spatial regions within a segment, the prefetch engine locates the next segment using the *next-seg* pointer. Additionally, we use the *num-insts* noted in the segment to determine how long after the Bundle starts executing the prefetch of the corresponding segment should be triggered. Specifically, prefetching for the $(N+1)_{th}$ segment is triggered when the number of instructions executed in the current Bundle surpasses the num-insts recorded for the N_{th} segment.

Core	
Architecture	Ice lake-like, ISA: x86-64, Freq.: 4GHz
BP Unit	L-TAGE [54] 64KB, ITTAGE [55] 64KB, BTB 8K entries, 8-way
Fetch	FTQ 24 entries, 16 bytes / cycle
LSU	Load 128 entries, Store 72 entries
ROB	352 entries
Memory Hierarchy	
Private L1 I-Cache	32 KB, 8 way, 2 cycles, 16 MSHRs, LRU
Private L1 D-Cache	48 KB, 12 way, 4 cycles, 16 MSHRs, LRU, NextLine Prefetcher
Private L2 Cache	512 KB, 8 way, 14 cycles, 32 MSHRs
Shared LLC	2 MB/core, 16 way, 50 cycles, 64 MSHRs
Memory	DDR4 2400MHz

Table 1. Parameters of the simulated processor.

Notably, the first and second segments are prefetched immediately at the start of each Bundle. This method ensures that prefetching is completed one segment ahead, largely keeping pace with execution to maintain good timeliness even when Bundles exceed the capacity of the L1-I cache.

Note that prefetching for a Bundle is triggered only upon commit of the triggering call/return instruction. Thus, Bundles are prefetched non-speculatively, which helps improve accuracy since the direction of the divergence points that delineate the Bundles may be difficult to predict. Although prefetching is triggered only when the Bundle starts executing, as explained above, the prefetcher rapidly runs ahead while its start-up latency is amortized over the long execution duration of the Bundle.

6 Methodology

6.1 Simulation Infrastructure

We conduct our evaluation using Gem5 [4, 18, 44]. Workloads are run using an out-of-order, execution-driven CPU model (O3CPU) in full-system simulation executing a full OS (Ubuntu). Our front-end baseline features FDIP, which is implemented in Gem5 using the open-sourced implementation from Ignite [52]. We also integrates a state-of-the-art indirect branch predictor ITTAGE [55] into gem5 based on Emissary's open-source implementation [45]. The workloads are first warmed up for 100 million instructions, following which statistics are collected over next 100 million instructions. The microarchitectural parameters for the modeled processor, resembling Intel Sunny Cove [28, 29], are listed in Table 1.

6.2 Benchmarks

We use 11 widely used server-side applications. They include three popular web-frameworks, beego, gin and echo from Web Frameworks Benchmark Suite [10], one HTTP/1-2-3 web server, Caddy [1] driven by the HTTP/2 benchmarking tool from nghttp2 [6], one graph database DGraph [2] and one ORM library gorm [5] from the ORM Benchmark Suite [7] (tested with PostgreSQL). In addition, for database/OLTP scenarios, we have conducted read-write tests using Sysbench [41] on TiDB and MySQL respectively. We further test TiDB driven by TPC-C [9], and mysql driven by ycsb [11] and sibench from OLTP-Bench [22].

6.3 Prefetch Mechanisms

We evaluate the following prefetchers:

- **Baseline (FDIP):** Implementation based on [52], featuring 24-entry fetch target queue. All of the evaluated prefetchers work alongside FDIP to maximize coverage and performance.
- **EFetch [21]:** SOTA caller-callee prefetcher, equipped with a 4K-entry callee predictor with single-cycle lookup latency; one entry is an ordered list of 3 callees, with 2 bit vectors for each callee. We calculated the signature using the top 3 items of the call stack, ensuring its effectiveness across all applications.
- **MANA [14]:** SOTA temporal prefetcher, equipped with a 4K-entry, 4-way index table with single-cycle lookup latency; the look-ahead depth is 3 spatial regions.
- **EIP [50]:** SOTA prefetcher and the winner of the IPC-1 instruction prefetching championship [3]. We use the balanced cost-performance configuration featuring a 4K-entry 8-way entangled table (40KB) and a 16-entry history buffer with an idealized single-cycle lookup latency for each component. Prefetches issued by FDIP are treated like demand accesses for the purposes of training EIP. This approach is found to perform better than ignoring FDIP-generated prefetches, which was confirmed to be the preferred approach by the authors of EIP.
- **Hierarchical (this work):** 16-entry Compression Buffer and a 512-entry (1.94KB) Metadata Address Table on chip. We faithfully simulate the latency and bandwidth of metadata memory access, which competes with regular traffic. The capacity of the Metadata Buffer is 512KB.

7 Evaluation

7.1 Performance Analysis

Figure 9 shows the relative IPC gains of all prefetchers across the benchmarks. MANA is affected by frequent resets of the core front-end [43] whenever any branch is mispredicted. When that happens, MANA has to stop prefetching and reindex the metadata to find the correct stream. As a result, it struggles to fully hide miss latency (as shown in Figure 2). A similar issue affects FDIP [52], which needs to flush the FTQ and restart prefetching from the same point as demand fetch any time a pipeline is flushed due to a branch misprediction. EFetch performs prefetching along the call-graph by leveraging the caller-callee relationships. However, like MANA, it also induces a trade-off between prefetch depth and prefetch accuracy as demonstrated in Sec 3.2 and Figure 2b. These

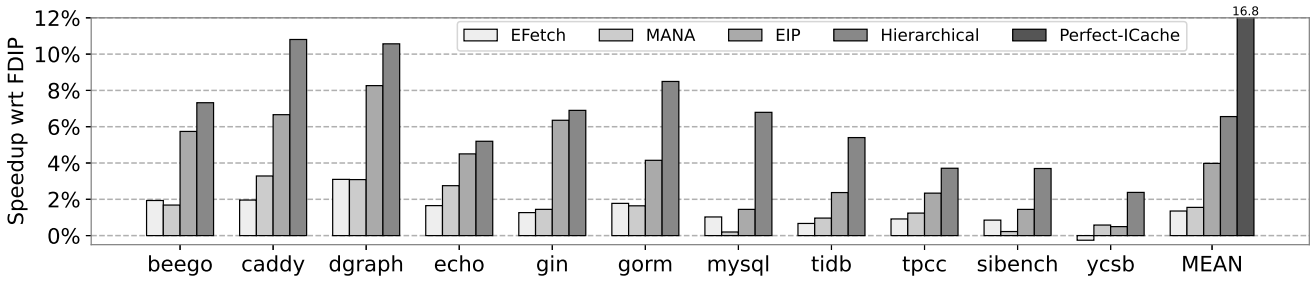


Figure 9. Performance improvement on top of FDIP.

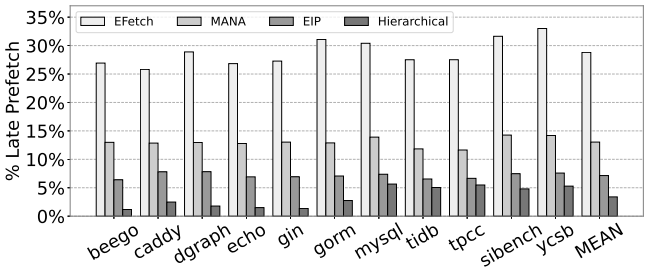


Figure 10. Percentage of prefetches arriving late.

issues fundamentally inhibit the prefetchers from hiding instruction stalls. As a result, MANA and EFetch manage only a small benefit over FDIP, achieving speedups of only 1.6% and 1.4%, respectively.

EIP strives to select triggers sufficiently in advance before the missed basic block, effectively eliminating miss latency through timely prefetching. For better timeliness, EIP operates at a greater distance compared to MANA or EFetch, which inevitably results in more control flow discrepancies between the trigger and prefetch target. Thus, while EIP improves prefetch timeliness (and, therefore, miss coverage), it sacrifices accuracy as discussed in Section 3.2, resulting in a performance improvement of 4.0%.

The highest performance is achieved by the proposed Hierarchical Prefetcher, with an average 6.6% IPC speedup over the FDIP baseline. These benefits are a result of high coverage and good timeliness stemming from the bulk approach to prefetching that is at the heart of the proposed technique.

Perfect L1-I: For perspective, we also compared Hierarchical Prefetching to a Perfect L1-I, which achieves an average IPC improvement of 16.8% over FDIP. Hierarchical Prefetching achieves 40% of that on average, and 77% in the best case.

7.2 Prefetch Timeliness

To assess the effect of coarse-grained and fine-grained prefetching on timeliness, we analyzed the proportion of late prefetches (hits in MSHR) across different methods, as illustrated in Figure 10. On average, 29% of prefetches in EFetch, 13% in MANA, 7% in EIP and only 3% in Hierarchical Prefetching are late.

Thanks to the large size of a Bundle, late prefetches only appear at the beginning of the Bundle with little effect on overall accuracy and coverage. We find that "cold start", where the prefetch latency is not hidden at the start of a Bundle's execution, comprises a few hundreds of cycles. By comparison, the average execution time of a Bundle exceeds tens of thousands of cycles. In short, the vast majority of prefetches arrive before the corresponding demand access.

The timeliness issue of MANA primarily stems from frequent resets when the actual control flow deviates from the prefetch path. Our tests indicate that it has an average prefetch distance of only 4 cache blocks, which prevents it from running far enough ahead to cover the long access latency of the LLC or main memory. Compared to MANA, EFetch faces a more severe timeliness problem due to its limitation of emitting only one callee deeper for each signature, a challenge that is difficult to overcome given hardware design and accuracy concerns. EIP outperforms EFetch and MANA in timeliness by learning miss latencies and prefetching in advance. However, it still falls short of achieving a prefetch depth that completely eliminates late prefetches, constrained by accuracy.

7.3 Prefetch Effectiveness

The extent to which a prefetcher can eliminate latency at a given cache level directly reflects its potential performance improvement. To that end, we study the latency observed by demand misses for instructions at each level of the cache for the various prefetching methods.

As shown in Figure 11, SOTA instruction prefetchers deliver little latency reduction on top of FDIP, which effectively explains their low performance improvement over FDIP. The best among evaluated prior techniques is EIP, which achieves an average latency reduction of 19.7% on top of FDIP. Hierarchical Prefetching reduces latency by 38.7% by eliminating 31.1% of L1 latency (others' best at 23.8%) and 52.2% of L2 latency (others' best at 18.7%). This demonstrates that Hierarchical Prefetching is effective at reducing demand access latency at both L1 and L2, the two main sources of miss latency.

According to our analysis, misses caused by accesses with long reuse distances are particularly important to cover as they tend to miss in the L2, exposing the high access latency

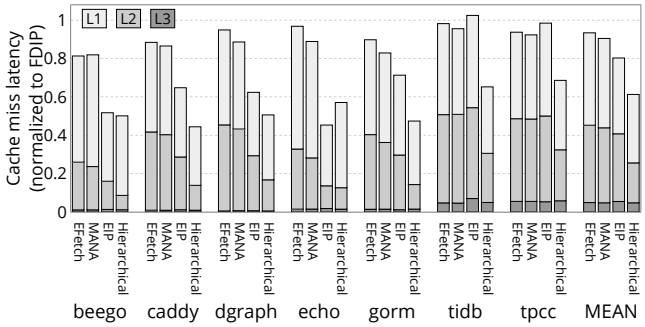


Figure 11. Cache miss latency for instructions. All prefetchers are on top of FDIP. MEAN shows the average of all 11 workloads.

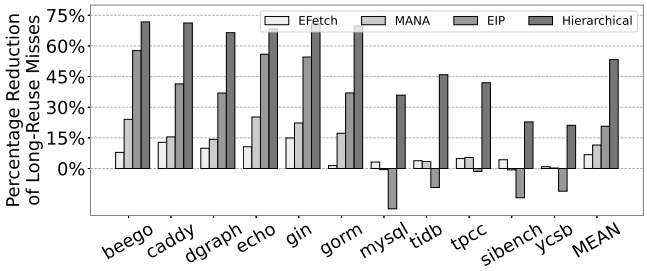


Figure 12. Effectiveness of eliminating L2 misses caused by top 10% long-range accesses over FDIP.

to the LLC. We find that these misses pose a particular challenge for SOTA prefetchers due to the high latency that they must hide. We study this phenomenon by first identifying the L2 misses from the 10% of instruction accesses with the longest reuse distances in terms of the number of unique interleaved cache lines with FDIP prefetching, which we term *long-range misses*. We then tested the effect of adding EFetch, MANA, EIP and Hierarchical prefetching to these misses.

As shown in the Figure 12, Hierarchical significantly outperforms prior prefetchers on long-range misses, eliminating an average of 53% and a peak of 72% of the misses, while EIP, EFetch and MANA average only 21%, 7% and 11%, respectively. This demonstrates that the proposed coarse-grained prefetching helps achieve better coverage on long-range misses, further explaining Hierarchical Prefetching's positive effect on L2 miss latency reduction depicted in Figure 11.

7.4 Prefetch Accuracy and Coverage

We assess the various prefetchers on prefetch accuracy and coverage, and correlate these measurements to the average prefetch distance of each prefetcher. Accuracy is computed as the percentage of prefetches that yield an L1-I hit for a demand fetch. Coverage is the percentage of demand misses converted into useful prefetches. Both accuracy and coverage are computed on top of the FDIP baseline (i.e., only the misses remaining after FDIP prefetching are considered).

Metric	EFetch	MANA	EIP	Hierarchical
Distance	3.4	4.3	6.1	**90**
Accuracy (L1-I)	**58%**	55%	30%	53%
Coverage (L1-I)	10%	14%	**48%**	37%
Coverage (L2)	8%	12%	23%	**54%**

Table 2. Average prefetch distance (in cache blocks), accuracy and coverage.

The prefetch distance is measured in cache blocks between the prefetch target and the trigger. Results are presented in Table 3.

Accuracy: For SOTA prefetchers, accuracy inversely correlates with prefetch distance, since larger prefetch distances are more likely to encounter a control flow divergence that compromises accuracy, resulting in accuracy ranging from 30% to 58% (as discussed in Section 3.2). We note that Hierarchical Prefetching operates at a much larger prefetch distance than prior techniques, owing to its coarse-grained prefetching strategy. Yet despite the large prefetch distance, Hierarchical achieves competitive accuracy, accomplished through two mechanisms: (1) delaying prefetching until a Bundle begins execution, which makes it non-speculative at a Bundle granularity, and (2) prefetching the cache blocks accessed during the Bundle's last execution, which makes it similar to temporal streaming within a Bundle.

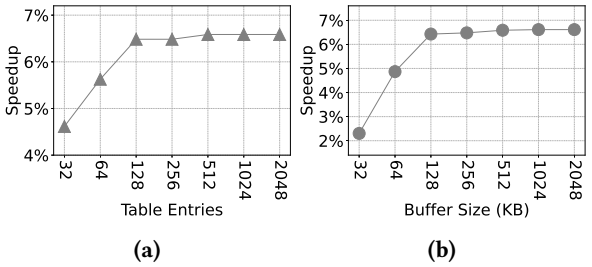
To better understand the accuracy differences between EIP and Hierarchical, we further analyzed the average number of branch targets or paths recorded per source for the prefetchers. Specifically, Hierarchical measures the number of distinct successors per basic block within a bundle, while EIP counts valid target basic blocks recorded for a source basic block. Across all workloads, for a given basic block, EIP will on average issue prefetches for 2.4 basic blocks to predict future paths, while Hierarchical issues prefetches for fewer than 1.5 basic blocks (when different basic blocks within the same bundle share the same successor basic block, Hierarchical only issues the corresponding prefetch once). When a single EIP trigger prefetches multiple paths, only one is likely to be executed, with the others often resulting in unused prefetches. This ultimately leads to EIP achieving a lower accuracy of 30%, compared to 53% of our method.

Coverage: With FDIP already being reasonably effective, SOTA prefetchers are unable to achieve much additional coverage because (like FDIP) they are susceptible to fine-grained control flow variations between the expected and actual access streams. As discussed in Section 3.2, such variations will inevitably result in a loss of coverage and timeliness (Figure 11). Specifically, EFetch and MANA achieve only 10% (8%) and 14% (12%) coverage for L1-I (L2), respectively. In contrast, by issuing prefetches for multiple potential paths when encountering hard-to-predict branches, EIP achieves higher coverage, with 48% for L1-I and 23% for L2. However, the large number of unused prefetches lead to eviction

of useful cache content, resulting in EIP having more L1-I demand misses than HP (despite EIP's higher coverage, as discussed in Section 7.3). In contrast, Hierchical Prefetching's coarse-grained approach to prefetching, whereby the entire recorded footprint of a Bundle is prefetched in bulk, achieves coverage of 37% at L1-I and 54% at L2, thus leading to high performance gains due to an effective reduction in misses for instructions.

We note that SOTA prefetchers achieve a lower coverage of misses at the L2 as compared to L1-I. L2 misses require a large prefetch distance to be covered effectively, and that presents a particular challenge for them as discussed previously. In contrast, Hierarchical Prefetching achieves a *higher* coverage at L2 than L1-I. The reason for that is that Bundles, which are formed statically, often have a dynamic instruction footprint that overflows the L1-I but is captured in the L2.

7.5 Sensitivity Analysis



(a) (b)

Figure 13. Average IPC speedups with (a) varying Metadata Address Table sizes and (b) varying Metadata Buffer sizes.

7.5.1 Metadata. We study the impact of scaling Metadata Buffer size and Metadata Address Table size on performance. Figure 13 shows the average performance improvement of Hierarchical Prefetching over the FDIP baseline under different configurations. The graphs justify our use of a Metadata Address Table with 512 entries and a 512KB Metadata Buffer, as larger configurations yield minimal further performance improvements.

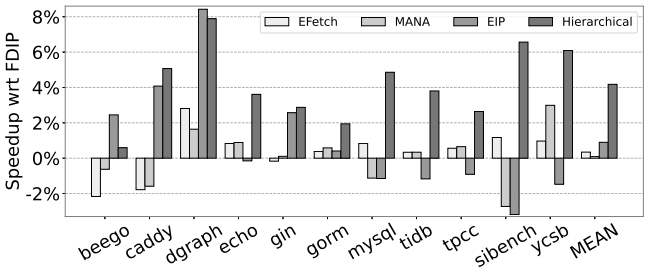
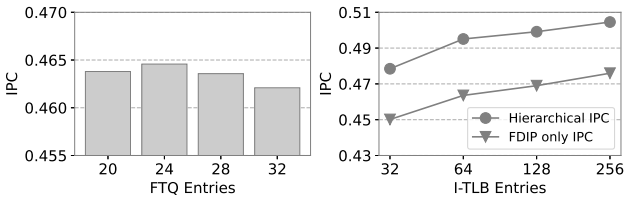


Figure 14. IPC speedups with infinite BTB capacity.

7.5.2 BTB. Then, we explored the effect of aggressive FDIP configurations on prefetching effectiveness. Previous work has shown that the main bottleneck of FDIP lies in the BTB capacity [25, 43, 57]. We thus investigated the effect of FDIP with infinite BTB capacity, not only to explore performance under an aggressive FDIP configurations but also to demonstrate that the differences between fine and coarse prefetching granularities stem from the mechanisms themselves, rather than merely metadata capacity.

As shown in Figure 14, when BTB capacity is sufficient, EFetch, MANA and EIP achieve only 0.3%, 0.1% and 0.9% average gains, respectively. In some applications, their performance is lower than that of FDIP alone. In contrast, Hierarchical is beneficial across all applications even in the presence of an aggressive FDIP configuration, averaging a gain of 4.2%.

This study points to a fundamental similarity and considerable overlap between fine-grained prefetching methods: when FDIP benefits from ample BTB capacity and consequently richer information to drive prefetching, it captures most of the gains typically associated with EFetch, MANA, or EIP, significantly limiting their potential for further performance enhancements. In contrast, Hierarchical Prefetching, with its robust ability to eliminate long-range misses and provide high prefetch timeliness, effectively addresses the performance shortfall of fine-grained techniques.



(a) (b)

Figure 15. IPC as a function of (a) FTQ size, (b) I-TLB size.

7.5.3 Fetch Target Queue. We tested FDIP across various FTQ sizes to identify the optimal configuration. As shown in Figure 15a, FDIP performs best with a 24-entry FTQ, with larger FTQs performing slightly worse. The trend that a deeper FTQ is mostly counter-productive is in agreement with the latest work on FTQ tuning [47].

7.5.4 I-TLB. We tested our method across various I-TLB capacities to explore how it interacts with address translation, as shown in Figure 15b. As expected, more I-TLB entries enhance the IPC for both baseline and Hierarchical due to fewer misses, with Hierarchical Prefetching consistently delivering over 6% IPC gains across all configurations.

7.5.5 L1-I. We evaluated all prefetching methods under varying L1-I cache sizes. Larger L1-I caches substantially improve EIP's accuracy, which increases from 30% at 32 KB to 42% at 256 KB. This accuracy improvement underscores that EIP is constrained by a high volume of unused prefetches, especially at small cache sizes, and the larger L1-I capacity can absorb some degree of pollution. HP also shows moderate

Prefetcher	L1-I Size	Accuracy	Coverage	Speedup
EFetch	32KB	58%	8%	1.4%
	64KB	56%	6%	1.1%
	128KB	56%	7%	1.1%
	256KB	53%	8%	1.3%
MANA	32KB	55%	14%	1.6%
	64KB	54%	12%	1.1%
	128KB	55%	12%	1.0%
	256KB	52%	11%	0.8%
EIP	32KB	30%	48%	4.0%
	64KB	32%	49%	2.6%
	128KB	38%	50%	2.3%
	256KB	42%	54%	2.0%
Hierarchical	32KB	53%	37%	6.6%
	64KB	57%	50%	5.3%
	128KB	59%	55%	5.7%
	256KB	57%	54%	5.1%

Table 3. Average prefetcher accuracy, coverage, and IPC speedup across all workloads under various L1-I cache sizes.

accuracy improvements (53% to 57%) as its coarse-grained prefetching strategy occasionally overflows the capacity of a small L1-I. As the L1-I size increases, both EIP and HP exhibit improved coverage because a larger L1-I can better accommodate a large volume of incoming prefetches. Meanwhile, IPC gains diminish with increasing L1-I sizes because the larger L1-I can inherently accommodate a larger instruction working set size, thereby reducing the benefit provided by instruction prefetchers. Nevertheless, Hierarchical prefetching retains a significant advantages (5.1% speedup at 256 KB L1-I) due to its strong ability to eliminate long-reuse-distance misses that even a large L1-I cannot capture.

7.6 Bundle Characteristics

Only a small fraction of the total functions are chosen as entry points for Bundles. Table 4 illustrates the proportion of functions labeled as Bundles across different applications, with an average of 3.7% and a maximum of 6.1%. The low proportion of functions that are the entry points of Bundles ensures that we can record all Bundle information with minimal overhead.

Bundles operate at a coarser granularity. The average recorded instruction footprint of Bundles ranges from 15KB to 68KB, approximately half to several times the size of the L1-I cache. When the bundle size exceeds the L1-I capacity, we prefetch by segments to roughly ensure that each group of prefetches fits well within it. In terms of execution time, Bundles execute for an average of 63045 cycles.

The instruction footprint of the Bundles exhibits high similarity across multiple executions. We calculated the average Jaccard index of all distinct Bundles encountered during execution of each benchmark. The Jaccard index for each Bundle is calculated as the average of Jaccard indices between all consecutive pairs of executions of that Bundle during the entire test, based on the set of cache blocks touched. Nearly all of our applications exhibit an average Jaccard index of over 0.8, with half exceeding 0.9. This demonstrates that the dynamic instruction footprint of Bundles exhibits a high degree of consistency and similarity, corroborating our method's ability to successfully capture coarse-grained repetition. This is also a crucial factor contributing to the effectiveness of our approach in terms of coverage and timeliness.

7.7 Memory Bandwidth

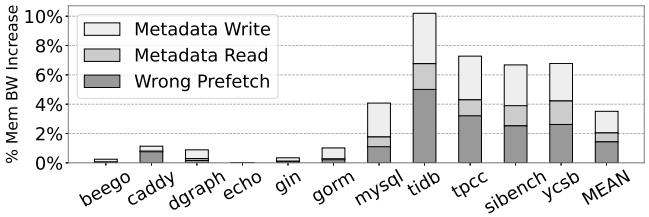


Figure 16. Memory bandwidth overhead.

Figure 16 shows the memory bandwidth usage of our method, normalized to the baseline. Memory bandwidth includes all memory accesses, covering instructions and data for both on-demand and prefetch operations. Hierarchical Prefetching's extra bandwidth overhead consists of overpredicted (i.e. unused) prefetch requests and metadata reads/writes associated with both recording and replaying.

As the figure shows, Hierarchical Prefetching introduces only a minor bandwidth overhead of 4% on average and 10% in the worst case. The fraction of this overhead due to overpredicted prefetches is 40%, with the remaining 60% due to metadata reads/writes.

7.8 Prefetching to L2

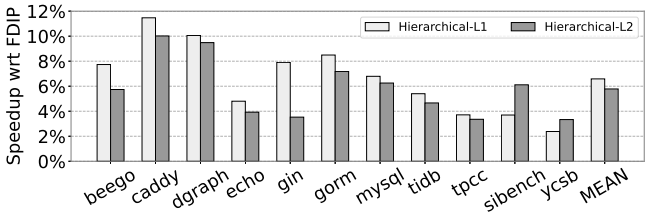


Figure 17. IPC speedups from L2 prefetching.

The coarse-grained approach of Hierarchical Prefetching makes it a good fit for prefetching directly into the L2, whose capacity can naturally accommodate even the largest Bundles. Direct prefetching into the L2 can thus avoid thrashing of the L1-I, improving bandwidth usage and energy efficiency. It is also synergistic with FDIP, which can cover short-range misses (to the L2) but struggles with long-range ones.

Benchmark	beego	caddy	dgraph	echo	gin	gorm	mysql	tidb	MEAN
Number of Static Bundles	1818	1674	4193	2800	1818	1913	3225	13446	3861
Total Functions	43772	61095	178190	45671	43785	44921	117616	475976	126378
Percentage of Bundles	4.15%	2.74%	2.35%	6.13%	4.15%	4.26%	2.74%	2.82%	3.67%
Average Footprint (KB)	57.03	33.42	48.07	63.08	68.22	36.45	15.28	17.55	42.39
Average Exe Cycles	54241	44187	92728	48007	78828	95120	72562	18690	63045
Average Jaccard Index	0.9668	0.9065	0.8102	0.9646	0.9371	0.8522	0.7978	0.8117	0.8809

Table 4. Bundle Statistics. The 3 bottom rows are per-Bundle averages.

We study the efficacy of prefetching into the L2 without any modifications to the Bundle formation algorithm, noting that in practice, tuning Bundle formation for L2 might yield better results. Results are presented in Figure 17. When directed to L2, Hierarchical Prefetching captures most of the benefits of prefetching to L1, yielding a performance improvement of 5.8% on average and 10% maximum. The good performance is attributed to effective elimination of long interval and high latency misses.

8 Conclusion

This paper introduces Hierarchical Prefetching, a software-hardware cooperative solution to excavate and utilize the coarse-grained application behavior to drive prefetching. We demonstrate that our method can eliminate the majority of long-range misses for instructions, reduce miss latencies throughout the cache hierarchy and deliver excellent prefetch timelines. These benefits combine to yield an average performance improvement of 6.6% on top of FDIP with only 1.94KB on-chip storage.

9 Acknowledgements

This work was supported by the National Natural Science Foundation of China Youth Fund under Grant No. 62202254, the National Natural Science Foundation of China under Grant No. 62250006, the Tsinghua University Initiative Scientific Research Program, the SuzhouTsinghua Innovation Leadership Program, and the Beijing National Research Center for Information Science and Technology. We would like to thank our shepherd, Prof. Heiner Litz, and the anonymous reviewers for their feedback. Youhui Zhang (zyh02@tsinghua.edu.cn) and Peng Qu (qp2018@mail.tsinghua.edu.cn) are the corresponding authors.

References

[1] [n. d.]. Caddy. https://caddyserver.com/.
[2] [n. d.]. Dgraph. https://dgraph.io/.
[3] [n. d.]. The First Instruction Prefetching Championship. https://research.ece.ncsu.edu/ipc/.
[4] [n. d.]. Gem5. https://github.com/gem5/gem5/tree/v23.0.1.0.
[5] [n. d.]. GORM. https://gorm.io/index.html.
[6] [n. d.]. Ngtttp2. https://nghttp2.org/.
[7] [n. d.]. ORM Bench. https://github.com/efectn/go-orm-benchmarks.
[8] [n. d.]. TiDB. https://www.pingcap.com/tidb/.
[9] [n. d.]. TPC-C. http://www.tpc.org/tpcc/.
[10] [n. d.]. Web Framework Bench. https://github.com/smallnest/go-web-framework-benchmark.
[11] [n. d.]. YCSB. https://github.com/brianfrankcooper/YCSB.
[12] Narasimha Adiga, James Bonanno, Adam Collura, Matthias Heizmann, Brian R Prasky, and Anthony Saporito. 2020. The ibm z15 high frequency mainframe branch predictor industrial product. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 27–39.
[13] Murali Annavaram, Jignesh M Patel, and Edward S Davidson. 2003. Call graph prefetching for database applications. *ACM Transactions on Computer Systems (TOCS)* 21, 4 (2003), 412–444.
[14] Ali Ansari, Fatemeh Golshan, Rahil Barati, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2022. Mana: Microarchitecting a temporal instruction prefetcher. *IEEE Trans. Comput.* 72, 3 (2022), 732–743.
[15] Islam Atta, Pinar Tözün, Anastasia Ailamaki, and Andreas Moshovos. 2012. Slicc: Self-assembly of instruction cache collectives for oltp workloads. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 188–198.
[16] Islam Atta, Pinar Tözün, Xin Tong, Anastasia Ailamaki, and Andreas Moshovos. 2013. STREX: boosting instruction cache reuse in OLTP workloads through stratified transaction execution. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*. 273–284.
[17] Grant Ayers, Nayana Prasad Nagendra, David I August, Hyoun Kyu Cho, Svilen Kanev, Christos Kozyrakis, Trivikram Krishnamurthy, Heiner Litz, Tipp Moseley, and Parthasarathy Ranganathan. 2019. Asmdb: understanding and mitigating front-end stalls in warehouse-scale computers. In *Proceedings of the 46th International Symposium on Computer Architecture*. 462–473.
[18] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH computer architecture news* 39, 2 (2011), 1–7.
[19] Ioana Burcea and Andreas Moshovos. 2009. Phantom-BTB: a virtualized branch target buffer design. *Acm Sigplan Notices* 44, 3 (2009), 313–324.
[20] Gino Chacon, Nathan Gober, Krishnendra Nathella, Paul V Gratz, and Daniel A Jiménez. 2023. A Characterization of the Effects of Software Instruction Prefetching on an Aggressive Front-end. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 61–70.
[21] Gaurav Chadha, Scott Mahlke, and Satish Narayanasamy. 2014. Efetch: optimizing instruction fetch for event-driven webapplications. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*. 75–86.
[22] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment* 7, 4 (2013), 277–288.
[23] Michael Ferdman, Cansu Kaynak, and Babak Falsafi. 2011. Proactive instruction fetch. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. 152–162.

[24] Michael Ferdman, Thomas F Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2008. Temporal instruction fetch streaming. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*. IEEE, 1–10.

[25] Bhargav Reddy Godala, Sankara Prasad Ramesh, Gilles A Pokam, Jared Stark, Andre Seznec, Dean Tullsen, and David I August. 2024. PDIP: Priority Directed Instruction Prefetching. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 846–861.

[26] Brian Grayson, Jeff Rupley, Gerald Zuraski Zuraski, Eric Quinnell, Daniel A Jiménez, Tarun Nakra, Paul Kitchin, Ryan Hensley, Edward Brekelbaum, Vikas Sinha, et al. 2020. Evolution of the samsung exynos cpu microarchitecture. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 40–51.

[27] Part Guide. 2011. Intel® 64 and ia-32 architectures software developer's manual. *Volume 3B: System programming Guide, Part* 2, 11 (2011), 1–64.

[28] Intel. [n. d.]. Ice Lake SP. https://www.intel.com/content/www/us/en/products/platforms/details/ice-lake-sp.html.

[29] Intel Ice Lake. [n. d.]. 7-Zip LZMA Benchmark. https://www.7-cpu.com/cpu/Ice_Lake.html.

[30] Yasuo Ishii, Jaekyu Lee, Krishnendra Nathella, and Dam Sunwoo. 2020. Rebasing instruction prefetching: An industry perspective. *IEEE Computer Architecture Letters* 19, 2 (2020), 147–150.

[31] Yasuo Ishii, Jaekyu Lee, Krishnendra Nathella, and Dam Sunwoo. 2021. Re-establishing fetch-directed instruction prefetching: An industry perspective. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 172–182.

[32] Paul Jaccard. 1912. The distribution of the flora in the alpine zone. 1. *New phytologist* 11, 2 (1912), 37–50.

[33] Prathmesh Kallurkar and Smruti R Sarangi. 2016. pTask: A smart prefetching scheme for OS intensive applications. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–12.

[34] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 158–169.

[35] Cansu Kaynak, Boris Grot, and Babak Falsafi. 2013. Shift: Shared history instruction fetch for lean-core server processors. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. 272–283.

[36] Cansu Kaynak, Boris Grot, and Babak Falsafi. 2015. Confluence: unified instruction supply for scale-out servers. In *Proceedings of the 48th International Symposium on Microarchitecture*. 166–177.

[37] Tanvir Ahmed Khan, Nathan Brown, Akshitha Sriraman, Niranjan K Soundararajan, Rakesh Kumar, Joseph Devietti, Sreenivas Subramoney, Gilles A Pokam, Heiner Litz, and Baris Kasikci. 2021. Twig: Profile-guided btb prefetching for data center applications. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 816–829.

[38] Tanvir Ahmed Khan, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. 2020. I-spy: Context-driven conditional instruction prefetching with coalescing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 146–159.

[39] Tanvir Ahmed Khan, Dexin Zhang, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. 2021. Ripple: Profile-guided instruction cache replacement for data center applications. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 734–747.

[40] Aasheesh Kolli, Ali Saidi, and Thomas F Wenisch. 2013. RDIP: Return-address-stack directed instruction prefetching. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. 260–271.

[41] Alexey Kopytov. 2004. Sysbench: a system performance benchmark. *http://sysbench. sourceforge. net/* (2004).

[42] Rakesh Kumar, Boris Grot, and Vijay Nagarajan. 2018. Blasting through the front-end bottleneck with shotgun. *ACM SIGPLAN Notices* 53, 2 (2018), 30–42.

[43] Rakesh Kumar, Cheng-Chieh Huang, Boris Grot, and Vijay Nagarajan. 2017. Boomerang: A metadata-free architecture for control flow delivery. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 493–504.

[44] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, et al. 2020. The gem5 simulator: Version 20.0+. *arXiv preprint arXiv:2007.03152* (2020).

[45] Nayana Prasad Nagendra, Bhargav Reddy Godala, Ishita Chaturvedi, Atmn Patel, Svilen Kanev, Tipp Moseley, Jared Stark, Gilles A Pokam, Simone Campanoni, and David I August. 2023. EMISSARY: Enhanced Miss Awareness Replacement Policy for L2 Instruction Caching. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–13.

[46] Tomoki Nakamura, Toru Koizumi, Yuya Degawa, Hidetsugu Irie, Shuichi Sakai, and Ryota Shioya. 2020. D-jolt: Distant jolt prefetcher. *The 1st Instruction Prefetching Championship (IPC1)* (2020).

[47] Surim Oh, Mingsheng Xu, Tanvir Ahmed Khan, Baris Kasikci, and Heiner Litz. 2024. UDP: Utility-Driven Fetch Directed Instruction Prefetching. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1188–1201.

[48] Andrea Pellegrini, Nigel Stephens, Magnus Bruce, Yasuo Ishii, Joseph Pusdesris, Abhishek Raja, Chris Abernathy, Jinson Koppanalil, Tushar Ringe, Ashok Tummala, Jamshed Jalal, Mark Werkheiser, and Anitha Kona. 2020. The Arm Neoverse N1 Platform: Building Blocks for the Next-Gen Cloud-to-Edge Infrastructure SoC. *IEEE Micro* 40, 2 (2020), 53–62. doi:10.1109/MM.2020.2972222

[49] Glenn Reinman, Brad Calder, and Todd Austin. 1999. Fetch directed instruction prefetching. In *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, 16–27.

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

[51] David Schall, Artemiy Margaritov, Dmitrii Ustiugov, Andreas Sandberg, and Boris Grot. 2022. Lukewarm serverless functions: characterization and optimization. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 757–770.

[52] David Schall, Andreas Sandberg, and Boris Grot. 2023. Warming Up a Cold Front-End with Ignite. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 254–267.

[53] David Seal. 2001. *ARM architecture reference manual*. Pearson Education.

[54] André Seznec. 2011. A 64 kbytes ISL-TAGE branch predictor. In *JWAC-2: Championship Branch Prediction*.

[55] André Seznec. 2011. A 64-Kbytes ITTAGE indirect branch predictor. In *JWAC-2: Championship Branch Prediction*.

[56] André Seznec. 2020. The fnl+ mma instruction cache prefetcher. In *IPC-1-First Instruction Prefetching Championship*. 1–5.

[57] Shixin Song, Tanvir Ahmed Khan, Sara Mahdizadeh Shahri, Akshitha Sriraman, Niranjan K Soundararajan, Sreenivas Subramoney, Daniel A Jiménez, Heiner Litz, and Baris Kasikci. 2022. Thermometer: profile-guided btb replacement for data center applications. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 742–756.