Appears in the First Instruction Prefetching Championship (IPC-1),
in conjunction with International Symposium on Computer Architecture (ISCA), 2020

MANA: Microarchitecting an Instruction Prefetcher

Ali Ansari[‡], Fatemeh Golshan[‡], Pejman Lotfi-Kamran[§], and Hamid Sarbazi-Azad[‡§]
[‡]Department of Computer Engineering, Sharif University of Technology
[§]School of Computer Science, Institute for Research in Fundamental Sciences (IPM)

***Abstract*—L1 instruction (L1-I) cache misses are a source of performance bottleneck when L1-I caches cannot hold the instruction footprint. Several instruction prefetchers, including Proactive Instruction Fetch (PIF), Return-Address-Stack Directed Instruction Prefetching (RDIP), and Shotgun were proposed to eliminate instruction cache misses as far as possible. While these proposals use entirely different mechanisms for instruction prefetching, we found that these proposals suffer from one or both of the following shortcomings: (1) a large number of metadata records to cover the potential, and (2) a high storage cost of each record. The first problem causes metadata-miss, and the second problem prohibits the prefetcher from storing enough records within reasonably-sized storage.**

In this paper, we make the case that the key to designing a powerful and cost-effective instruction prefetcher is choosing the right metadata record and microarchitecting the prefetcher to minimize the storage cost. We find that high spatial correlation among instruction accesses leads to compact, accurate, and minimal metadata records. We also show that chaining these spatial records is an effective manner to enable robust and timely prefetching. Based on the findings, we propose *MANA*, which outperforms the competitors when all of them use a 128 KB storage budget. Moreover, we show that MANA considerably outperforms the other approaches when a limited storage budget is provided for the prefetchers that makes MANA a more practical solution to be implemented in real-world processors.

***Index Terms*—Caching, Instruction Cache Misses, Temporal Prefetching**

I. Introduction

Instruction cache misses are a well-known source of performance degradation when the limited-capacity L1 instruction (L1-I) cache cannot capture a large number of instruction blocks demanded by a processor [1]–[3]. Instruction prefetching is a technique to address this problem. The most common instruction prefetchers are sequential prefetchers that, upon activation, send some prefetch requests for a small number of subsequent blocks [4], [5]. However, prior work has shown that sequential prefetchers leave a significant fraction of instruction misses uncovered, and hence, there is a substantial opportunity for improvement [1], [3], [5], [6].

Sequential prefetchers' limitations motivated researchers to propose more sophisticated prefetchers. Proactive Instruction Fetch (PIF) is a pioneer that showed a hardware instruction prefetcher could eliminate most of the instruction cache misses [1]. However, the proposed prefetcher is impractical because it requires over 200 KB storage cost per-core compared to the baseline design.

The follow up work tried to reduce the required storage cost of an instruction prefetcher. For example, Return-Address-Stack Directed Instruction Prefetcher (RDIP) [3] takes advantage of the content of the return-address-stack (RAS) for prefetching, and hence, lowers the per-core storage cost to over 60 KB. The latest proposal, Shotgun [2], offers instruction prefetching with a negligible storage cost by tracking the control flow changes that are already recorded in a branch target buffer (BTB).

In this paper, we evaluate RDIP, Shotgun, and PIF, as the three state-of-the-art prefetchers that use entirely different approaches for instruction prefetching and show they are not the final solution for instruction prefetching since they require high storage cost to cover the available potential. Moreover, we make a case for carefully choosing a prefetcher's metadata to minimize the metadata storage as well as microarchitecting the metadata storage to further reduce the storage overhead. We introduce *MANA* prefetcher that offers 26.6% speedup when a 128 KB storage budget is provided for it. Moreover, it offers 23.8% speedup when we limit its storage to only 16 KB. In this case, MANA outperforms the state-of-the-art prior work with a large gap. This small storage requirement empowers MANA to prefetch for smaller L1-I caches to eliminate the storage overhead as compared to the baseline design. We show that MANA offers a considerable performance gain with a 16 KB L1-I cache as compared to the competing prefetchers with a 32 KB cache.

II. Background

A. Temporal Prefetchers

Temporal prefetching is based on the fact that the sequence of instruction cache accesses or misses is repetitive, and hence, predictable [1], [7]. Consequently, temporal instruction prefetchers record and replay the sequence to eliminate future instruction cache misses. PIF [1] is the state-of-the-art temporal prefetcher that uses the sequence of retire-order instruction stream to prefetch for the L1-I cache. The main shortcoming of temporal prefetchers is their high storage budget requirements. As an example, PIF requires more than 200 KB storage budget per-core to work effectively.

B. RAS Directed Instruction Prefetcher (RDIP)

Return-Address-Stack Directed Instruction Prefetcher (RDIP) [3] observes that the current state of the return-address-stack (RAS) can give an accurate representation of the program's state. To exploit this observation, RDIP XORs the four top entries of the RAS and calls it a *signature*. Then it assigns the observed instruction cache misses to the corresponding signature. Finally, it stores these misses in a set-associative table named Miss Table that is looked up using the signature. RDIP reduces the per-core storage to

over 60 KB. While RDIP requires considerably lower storage as compared to PIF, it still needs a significant storage budget.

C. BTB-Directed Prefetchers

BTB-directed prefetchers are advertised as metadata-free prefetchers. Fetch Directed Instruction Prefetcher (FDIP) is the pioneer of such prefetchers [8]. The main idea is to decouple the fetch engine from the branch predictor unit. This way, the branch predictor unit goes ahead of the fetch stream to discover the instruction blocks that will be demanded shortly. Then the prefetcher checks if any of those blocks are missing and prefetches them.

The main bottleneck of BTB-directed prefetchers is BTB misses [2], [9]. Shotgun [2] is the state-of-the-art BTB-directed instruction prefetcher that proposed a new BTB organization to mitigate the BTB miss problem. However, as we will show in this paper, this problem is still the factor limiting its efficiency.

III. Motivation

A. Performance Comparison

Figure 1 compares the performance improvement of RDIP, Shotgun, and PIF over a baseline without a prefetcher. For RDIP and Shotgun, along with the authors-proposed configuration, we evaluate a configuration with infinite storage. Moreover, we evaluate an implementation of PIF in which the history buffer has 4 K entries while it has 32 K entries in the authors-proposed configuration. Results reveal three essential facts. First, PIF outperforms RDIP and Shotgun with a large gap. It means that the reduction in storage in RDIP and Shotgun is achieved at the cost of losing considerable speedup. Second, RDIP and Shotgun fill this gap when infinite storage is available to them. Consequently, the considerable performance gap in the original configuration is because RDIP and Shotgun are incapable of holding the large number of records that they require to prefetch effectively. Finally, PIF loses performance when the history buffer size is decreased. It means that having such a long history buffer is essential for PIF to exploit the potential.

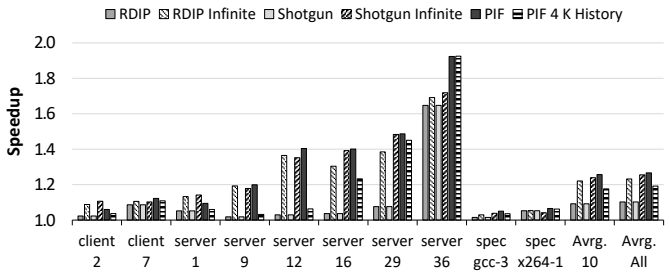

Fig. 1. Speedup offered by state-of-the-art prefetchers.

B. Number of Prefetching Records

To find out why RDIP and Shotgun suffer from metadata-miss, we should know what their prefetching records are and how they are stored. RDIP creates signatures that are the bit-wise XOR of the four top entries in the RAS. The signatures are used to look up the Miss Table in which the addresses of missed blocks are recorded. As a result, RDIP should have a record for each observed signature in the Miss Table, and its number of required records is equal to the number of distinct signatures. We note that RDIP suggested a 4 K-entry Miss Table that is organized as a 4-way set-associative structure.

On the other hand, Shotgun needs to store basic blocks in its BTBs. As a result, the prefetching record of Shotgun is a basic block, and BTB should be large enough to accommodate the basic blocks. To hold these basic blocks, Shotgun uses three BTBs that hold 2 K entries altogether. However, Shotgun attempts to prefill its BTBs to compensate for its relatively small BTBs. Nevertheless, Figure 1 shows that even with the prefilling mechanism, the metadata-miss problem is still a considerable bottleneck.

Finally, PIF benefits from spatial regions. Each spatial region consists of a block address, called a *trigger*, and a footprint that shows the bitmap of accessed blocks around the trigger. In consequence, spatial regions are the prefetching records that PIF should successfully hold. PIF writes these spatial regions in a 32 K entry circular history. Moreover, it uses an 8 K entry index table that records the latest appearance of a specific spatial region in which entry of the circular history is recorded.

Figure 2 shows the number of distinct records that are observed for each prefetcher. It can easily be inferred that RDIP and Shotgun have a significantly larger number of distinct prefetching records that cannot be held in their Miss Table and BTBs, respectively. Moreover, we observe that PIF has a significantly smaller number of distinct records. The absolute value is less than 4 K on average, and an 8 K-entry index table can accommodate the records.

While Figure 2 suggests that PIF has fewer distinct prefetching records, its design cannot exploit this advantage. Figure 1 shows that by decreasing the number of history-buffer entries from 32 K to 4 K, the obtained speedup shrinks from 26% to 20%. The reason is that multiple instances of a spatial-region record may be written in PIF's history buffer. Consequently, while the number of distinct records is less than 4 K, an appropriate history buffer should be much larger to hold all of the records successfully.

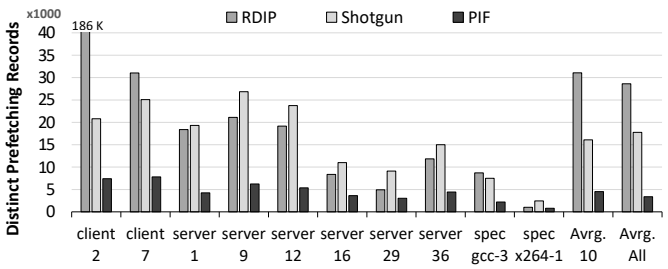

Fig. 2. Number of distinct prefetching records.

IV. MANA Prefetcher

In this section, we describe how MANA prefetcher works. Abstractly, MANA creates the spatial regions using a spatial region creator and stores them in a set-associative table named MANA_Table. Each spatial region is also associated with a pointer to another MANA_Table entry in which its successor is recorded. To reduce the storage cost, MANA exploits this

observation that there are a small number of distinct high-order-bits patterns. Consequently, instead of recording the complete trigger address that is the largest field of spatial regions, MANA uses the pointers to the observed high-order-bits patterns that need a considerably fewer number of bits.

A. Recording

1) Spatial Region Creator: Spatial Region Creator (SRC), is responsible for creating MANA's prefetching records. Spatial regions consist of a trigger address and a footprint that shows which instruction blocks are observed in the neighborhood of the trigger address. SRC tracks the demanded instruction stream and extracts its instruction blocks. If the current instruction block is different from the last observed instruction block, SRC attempts to assign this new instruction block to a spatial region. SRC has a queue of spatial regions named Spatial Region Queue (SRQ). After detecting a new instruction block, SRC compares this instruction block with SRQ entries. If this block falls in the address space that is covered by one of the SRQ entries, SRC sets the corresponding bit in that spatial-region footprint. Otherwise, SRC dequeues an item from SRQ, creates a new spatial region whose trigger address is the new instruction block, resets the footprint, and enqueues it in the SRQ.

2) MANA_Table: When SRC dequeues an entry from SRQ to enqueue a new spatial region, the evicted spatial region is inserted into MANA_Table. MANA_Table stores the spatial regions and uses a set-associative structure with Least Recently Used (LRU) replacement policy that is looked up by the trigger address of the spatial region. Upon an insertion, if a spatial region hit occurs, the spatial region's footprint is updated with the latest footprint. Otherwise, the LRU entry is evicted, and the new spatial region is inserted into MANA_Table.

3) Finding the Next Spatial Region: We include in MANA_Table's prefetching record a pointer to another entry of MANA_Table. Whenever a spatial region is inserted into MANA_Table, MANA records its location. By knowing this location, when MANA records a new entry in the table, the pointer of the last recorded spatial region is set to the location of the new entry. Using these pointers, MANA can chase the spatial regions one after another by iteratively going from a spatial region to its successor.

4) High-Order-Bits Patterns: Considering a 64-bit address space[1] and a 4 K-entry 4-way set-associative MANA_Table, each record requires a 48-bit trigger-address tag, an 8-bit footprint, and a 12-bit pointer to the successor. The 8-bit footprint is derived from prior work [1], [2], [12]. To further reduce the storage cost, we observe that there is a considerable similarity between the high-order bits of the instruction blocks, and there are a few distinct patterns due to the high spatial locality of the code base of programs. As a result, we divide the trigger address tag into two separate parts, a partial tag, and the rest of the high-order bits.

[1]We assume a 64-bit address space because of the restrictions in the IPC-1 [10]; however, the actual physical address space requires fewer bits [11].

We store the partial tag in MANA_Table and the rest of the bits in a separate structure. The division of tag should be done in a way to minimize the storage overhead. If we devote more bits for the partial tag, we will have fewer high-order-bits patterns (HOBPs), but we need to store longer partial tags in MANA_Table. On the contrary, if we devote a smaller number of bits to the partial tag field, we will encounter more distinct HOBPs.

MANA stores HOBPs in a set-associative table named high-order-bits patterns' table (HOBPT). Every new observed HOBP is inserted in HOBPT. Moreover, each MANA_Table record has a *HOBP index*, which points to a HOBPT entry in which the corresponding HOBP is recorded.

B. Replaying

MANA uses the recorded spatial regions to prefetch for the L1-I cache. For this purpose, MANA takes advantage of the stream address buffer (SAB) that is previously used by prior temporal prefetchers [1], [12], [13]. SAB is a fixed-length sequence of spatial regions that are created by chasing the spatial regions one after another from the pointers that are stored in MANA_Table. Moreover, SAB has a pointer to the MANA_Table entry that the last spatial region is fetched from and inserted into SAB.

MANA uses SABs to prefetch using the following procedure. MANA attempts to have a fixed lookahead ahead of the fetch stream to issue timely prefetches. This lookahead is defined as the number of spatial regions that MANA prefetches ahead when it observes an instruction cache block. MANA tracks the fetch stream and extracts its instruction block addresses. If the block address falls in the address space that is covered by a spatial region in a SAB, MANA checks the number of spatial regions that are prefetched after the matched spatial region, and hence, are inserted into SAB. If this number is lower than the lookahead, MANA chases the spatial regions using SAB's pointer to MANA_Table to have sufficient lookahead. If MANA finds no SAB associated with the block address, it considers the instruction block as the trigger address of a spatial region and looks up MANA_Table to find the corresponding spatial region. If MANA_Table finds a match, MANA evicts the LRU SAB entry (if it has multiple SABs) and creates a new SAB by inserting the found spatial region into SAB and chasing its successor pointer to find the next spatial region. MANA repeats this process until the number of inserted spatial regions into SAB reaches the predefined lookahead depth. Finally, MANA extracts the instruction blocks that are encoded in the footprint of the inserted spatial regions and prefetches them.

C. Multiple Successors

MANA_Table can record a single successor for each spatial region. However, some spatial regions may have multiple successors because they may be called from different points. To take care of such spatial regions, we divide MANA_Table to two distinct tables. MANA_Table (single) holds a single pointer to the successor region. Once MANA finds that the new successor of a spatial regions is not the last recorded one, it moves this spatial region to the MANA_Table (multiple) that holds multiple pointers to the successor spatial regions.

These pointers are organized as a circular history. For pointer chasing, MANA finds the last inserted successor into the history, then finds the second last insertion of that successor, and chases its subsequent entry in the circular history.

D. Storage Requirements

We have carefully analyzed different configurations for MANA to propose the best performing one. We find that a 2-bit partial tag appropriately solves the trade-off between the storage requirements of the HOBPT and MANA_Table. Moreover, a 128 entry 8-way set-associative HOBPT is sufficient to hold all observed HOBPs. Consequently, HOBP index requires seven bits. Footprints are 8-bit long that show which instruction blocks are accessed in the eight blocks ahead of the trigger block. We set the lookahead to three, SRQ size to 8, and use a single SAB that tracks 5 spatial regions. If we use this configuration along with a 4 K entry MANA_Table that holds a single successor pointer, this configuration requires 16.3 KB storage cost that will be evaluated in Section VI. However, to exploit all 128 KB storage budget that is given in the IPC-1 [10], we use a 1 K entry, 8-way set-associative HOBPT that is 5 KB. Moreover, we utilize a 16 K entry, 4-way set-associative MANA_Table (single) along with a 4 K entry, 4-way set-associative MANA_Table (multiple) that require 74 and 43 K, respectively. SRQ and SAB need 0.1 KB altogether. Finally, we use a 64-entry prefetch queue to hold the prefetch candidates that needs 0.45 KB. Consequently, the whole storage cost of this MANA configuration is 122.55 KB that fits in the given storage budget.

V. Methodology

To evaluate our proposal, we use the simulation framework provided by the first instruction prefetching championship (IPC-1) [10]. We use ChampSim [14] simulator with the configurations provided by IPC-1 [10]. Each benchmark is executed for 50 million instructions to warm up the system, including the caches, the branch predictor, and prefetchers' metadata. The subsequent 50 million instructions are used to collect the evaluation metrics, including Instruction Per Cycle (IPC).

We use public benchmarks that are provided by IPC-1 [10]. While we execute all 50 benchmarks, as it is not possible to show all the results, we selected ten benchmarks that represent various observed behaviors. Moreover, we report the average of the ten selected benchmarks as well as the average of all 50 benchmarks.

We compare our prefetcher, MANA, with RDIP [3], Shotgun [2], and PIF [1] to show how much these proposals are effective to eliminate the instruction cache misses.

VI. Evaluation Results

A. Performance Improvement

We evaluate the obtained speedup by competing prefetchers when we provide 128 KB storage budget for the prefetchers according to the competition rules as well as when we limit their storage budget to 8 KB and 16 KB to represent their behavior when a more practical storage budget is provided for them. Figure 3 shows that when we exploit all 128 KB storage budget, Shotgun, PIF, and MANA offer near the same level of performance. However, MANA outperforms both of them by providing 26.6% speedup on top of the baseline. When we limit the storage budget to 8 and 16 KB, we see that MANA has a large gap with its competitors offering 18.7 and 23.8% speedup on top of the baseline. Its clearly an advantage for MANA that covers a considerable part of the available potential with a small storage requirement that makes it a suitable design to be used in real-world processors.

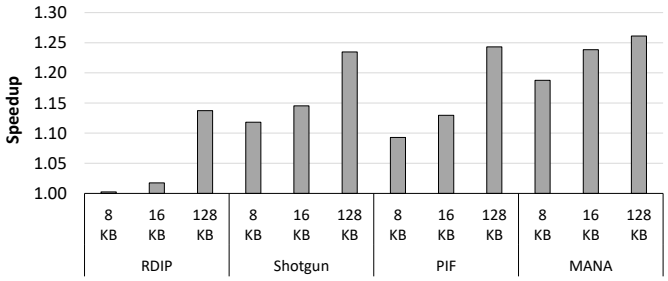

Fig. 3. Speedup of competing prefetchers with various storage budgets.

B. MANA with a Smaller Cache Size

MANA's good speedup with a small storage budget motivates us to use it to prefetch for smaller cache sizes to completely eliminate the storage cost compared to the baseline design. Figure 4 shows that when we decrease the L1-I cache size and use MANA prefetcher with 16 KB storage budget to prefetch for it, MANA still offers good speedup. Speedup is 22% and 20% for 16 KB and 8 KB caches, respectively. Note that MANA is designed to be independent of what is going on L1-I caches. In other words, MANA does the same independent of the L1-I cache. The offered speedup on 16 KB cache is very close to the speedup obtained by the conventional 32 KB cache. So, we can use MANA with a 16 KB cache to avoid storage overhead. This way, the design imposes no storage overhead while offers almost the same performance as MANA with a 32 KB cache. We have also evaluated the impact of cache size reduction on the external bandwidth usage. However, the external bandwidth usage increases proportionally to the L1-I capacity reduction factor. The external bandwidth usage of 16 K and 8 K L1-I caches increases by 2.3× and 3.4×, respectively. While the external bandwidth increases, as it is the bandwidth between L1 and L2 within the chip, depending on the specifics of processors, it may be beneficial to trade it for significant performance improvement.

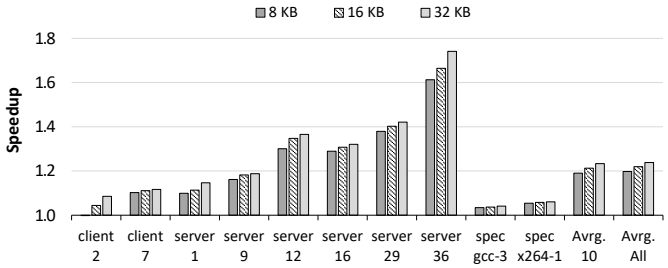

Fig. 4. Speedup of MANA for various L1-I cache sizes.

REFERENCES

[1] M. Ferdman, C. Kaynak, and B. Falsafi, "Proactive Instruction Fetch," in *Proceedings of the 44th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, Dec. 2011, pp. 152–162.

[2] R. Kumar, B. Grot, and V. Nagarajan, "Blasting Through the Front-End Bottleneck with Shotgun," in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2018, pp. 30–42.

[3] A. Kolli, A. Saidi, and T. F. Wenisch, "RDIP: Return-Address-Stack Directed Instruction Prefetching," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2013.

[4] A. J. Smith, "Sequential Program Prefetching in Memory Hierarchies," *Computer*, vol. 11, no. 12, pp. 7–21, Dec. 1978.

[5] A. Ansari, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Divide and Conquer Frontend Bottleneck," in *Proceedings of the 47th Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2020.

[6] M. Bakhshalipour, P. Lotfi-Kamran, A. Mazloumi, F. Samandi, M. Naderan-Tahan, M. Modarressi, and H. Sarbazi-Azad, "Fast data delivery for many-core processors," *IEEE Transactions on Computers*, vol. 67, no. 10, pp. 1416–1429, 2018.

[7] M. Ferdman, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Temporal Instruction Fetch Streaming," in *Proceedings of the 41th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, Nov. 2008, pp. 1–10.

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

[9] R. Kumar, C.-C. Huang, B. Grot, and V. Nagarajan, "Boomerang: A Metadata-Free Architecture for Control Flow Delivery," in *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2017, pp. 493–504.

[10] "IPC-1," https://research.ece.ncsu.edu/ipc/, 2020.

[11] "RAM limit," https://en.wikipedia.org/wiki/RAM_limit/, 2020.

[12] C. Kaynak, B. Grot, and B. Falsafi, "SHIFT: Shared History Instruction Fetch for Lean-core Server Processors," in *Proceedings of the 46th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, Dec. 2013, pp. 272–283.

[13] ——, "Confluence: Unified Instruction Supply for Scale-out Servers," in *Proceedings of the 48th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, Dec. 2015, pp. 166–177.

[14] "ChampSim," https://github.com/ChampSim/, 2020.