Appears in the First Instruction Prefetching Championship (IPC-1),
in conjunction with International Symposium on Computer Architecture (ISCA), 2020
IPC-1 Submission **#92** – Confidential Draft – Do NOT Distribute!!

Divide and Conquer Instruction Cache Misses

***Abstract*—L1 instruction cache (L1i) misses are a major source of performance degradation when the instruction footprint cannot be captured in the L1i. Sequential prefetchers like a next-line prefetcher are a common solution to mitigate this problem. However, this approach falls short of efficiency when the program has lots of complex control flow changes that occur frequently. This observation has motivated researchers to suggest a myriad of sophisticated proposals to address this problem. However, we find that there is still significant room for improvement. Hence, in this paper, we introduce a new instruction prefetcher to exploit the available potential.**

In this paper, we address the L1i cache miss problem using a divide and conquer approach. We carefully analyze why an instruction cache miss occurs and how it can be eliminated. We divide instruction cache misses into sequential and discontinuity misses. A sequential miss is a cache miss that is spatially right after the last accessed block, and the remaining misses are discontinuities.

While sequential and discontinuity prefetchers are already proposed, in this paper, we show that the conventional implementation of these prefetchers cannot adequately cover the misses because of their shortcomings. Accordingly, we recommend an enhanced implementation of each prefetcher. We find that for a sequential prefetcher, there is a trade-off between timeliness and accuracy. Consequently, we propose *SN4L+wrong_SNL* prefetcher that attempts to provide both accurate and timely prefetches. Moreover, a conventional discontinuity prefetcher uses a single discontinuity target for each record, and as such, its lookahead is limited to a single discontinuity ahead of the execution stream, which limits its efficiency. On top of that, it records an instruction block address per record that results in a considerable storage cost. We introduce *Dis* prefetcher to address the shortcomings. Our proposal offers 25% speedup as compared to the baseline without any prefetcher when 128 KB storage budget is provided for it and outperforms the state-of-the-art prefetcher by 3% when a small 8 KB storage budget is provided to it.

I. Introduction

Frequent L1 instruction (L1i) cache misses are an important source of performance degradation when the L1i cache is incapable of capturing the large number of instruction blocks that are required [3], [4], [7], [8]. Upon an L1i miss, no instruction is fetched until the missing block arrives from the lower levels of the memory hierarchy. In consequence, no new instruction is fed into the core to be executed, which eventually stalls the core itself. Consequently, instruction cache misses result in a considerable performance loss.

Instruction prefetching is the general solution to address this problem. The simplest and most common instruction prefetchers are sequential prefetchers that issue prefetch requests for a number of subsequent blocks when they are triggered [10]. Sequential prefetchers are mainly distinguished by the number of prefetch requests that they issue upon activation, which is called the prefetch depth. Higher depths usually offer more timely prefetches by prefetching blocks that are sufficiently far ahead of the current demanded block. However, this may impose lots of useless prefetches that pollute the cache, queues, and the network-on-chip. This pollution may offset the benefits of better timeliness. As a result, there is a trade-off between accuracy and timeliness when a fixed prefetch depth is assigned to a sequential prefetcher.

Sequential prefetchers only eliminate sequential misses. However, in modern workloads, branch instructions frequently change the control flow. These changes may access a cache block that is not in the cache, and consequently, a discontinuity cache miss occurs. As sequential prefetchers cannot cover such misses, it is necessary to have a more powerful prefetcher that can take care of these misses as well.

Discontinuity prefetcher (DP) is a simple approach to assist sequential prefetchers [11]. DP uses the next-4-line prefetcher to cover sequential misses. Every time an instruction miss occurs, DP creates a record that includes the *source* and *target* addresses, which are the last hit block and the missed block, respectively. The pair of source and target blocks is called a *discontinuity*. Then DP stores the discontinuity in the *discontinuity table*. This table is looked up by the source address, and the target is recorded in the corresponding entry. As a result, the discontinuity table is direct-mapped and tagless. When the sequential prefetcher looks up the cache to decide whether it should send a sequential prefetch or not, DP looks up the subsequent blocks in the discontinuity table. If a valid target is found for a source address, a discontinuity prefetch request will be sent for the target.

DP has several shortcomings. First, to reduce the storage requirements, DP does not record the source address in the table. Consequently, the discontinuity table is organized in a direct-mapped and tagless structure. Consequently, conflicts cannot be detected in such a table, which may result in inaccurate prefetching. Moreover, DP uses a single target for each source address. However, some blocks may have multiple targets because of having multiple branches in the source block. As such, DP cannot cover these misses. Moreover, the discontinuity prefetcher is limited to prefetching a single discontinuity ahead of the demanded block. As a result, it cannot discover the blocks far ahead of the demanded block to issue timely prefetches, especially in the case of having frequent discontinuities that occur shortly one after another. These shortcomings of DP motivated the researchers to develop other techniques to tackle the instruction cache misses.

Many proposals suggested temporal prefetching [3], [4]. In temporal prefetching, the sequence of observed cache accesses [3] or misses [4] is recorded and replayed to cover the misses. Proactive Instruction Fetch (PIF) is the state-of-the-art temporal prefetcher that offers a significant performance improvement at the cost of imposing over 200 KB storage cost per core to the baseline design [3]. Other prefetching techniques offered proposals to mitigate the storage cost. Return-Address-Stack Directed Instruction Prefetcher (RDIP) [6] uses the content of the Return-Address-Stack (RAS) to detect the program context and prefetch the missed blocks that are already observed for that context. RDIP reduces the storage cost to over 60 KB. However, BTB-directed prefetchers impose much lower storage cost by prefetching according to the control flow changes that are already recorded in the branch target buffer (BTB), which is available in the conventional processors [7]–[9]. Nevertheless, prior work has shown that BTB misses are a fundamental bottleneck for this prefetching scheme [7], [8]. Shotgun is the state-of-the-art BTB-directed prefetcher that requires only 6 KB storage to work, assuming that the processor has a 2 K-entry BTB [7].

In this paper, we introduce a new prefetcher based on the sequential and discontinuity prefetchers. We show that our prefetcher offers the level of performance that is offered by the state-of-the-art temporal and BTB-directed prefetchers when a considerable storage budget is provided for them. Moreover, we

show that our prefetcher offers a significant speedup even when a small storage budget is provided to the prefetcher and outperforms the state-of-the-art prefetchers with a large gap.

II. SN4L+wrong_SNL+Dis Prefetcher

SN4L+wrong_SNL+Dis prefetcher is a combination of two sequential instruction prefetchers, named SN4L and wrong_SNL, and a discontinuity prefetcher named Dis prefetcher. This section presents the details of these prefetchers.

A. Selective Next-Four-Line Prefetcher (SN4L)

Selective-next-four-line (SN4L) prefetcher is proposed to provide both accurate and timely sequential prefetches. For each instruction block, SN4L devotes a single bit named *status* bit. If this bit is set for a block, it means the state of that block is *demanded*; otherwise, the block is *prefetched*. SN4L issues a prefetch request for a block if the state of that block is *demanded*.

Whenever a block is demanded by the processor, its status bit is set, and every time SN4L prefetches a block, this bit is reset. When SN4L is triggered to issue some prefetches, it checks the status bits of the next four subsequent blocks and issues prefetches just for those blocks that are demanded. In this way, SN4L issues timely prefetches, and at the same time, removes redundant or useless prefetches.

SN4L stores the status bits of the blocks in a large, tagless, and direct-mapped table named *SeqTable* in which an entry is a status bit. This decision helps SN4L to increase the number of SeqTable's entries to reduce the conflicts. We have used a 64 K entry SeqTable for SN4L that imposes 8 KB storage overhead.

B. Recently Looked Up (RLU) Filtering

An aggressive prefetching mechanism like SN4L may still impose a large number of redundant requests. Consider the following sequence of demanded blocks:

$$...,(A,\ A{+}1)_n,...$$

In this sequence, a backward branch in block $A+1$ brings the execution back to block A. As $A+1$ is demanded, SN4L will attempt to prefetch $A+1$ when it is triggered on block A. However, it is hit, and such a prefetch is unnecessary. To avoid such cases, we keep track of the blocks that are recently prefetched and filter the prefetch candidates that are recently prefetched. To do so, we use a first-in-first-out structure named *RLU*. Every time a prefetch request is sent for a block, it is also pushed into RLU. The prefetcher only sends a prefetch request for a block if it is not in RLU. RLU filtering is not only used for SN4L prefetcher but also for the rest of the prefetchers that we introduce.

RLU filtering has some advantages. It saves precious on-chip bandwidth to look up the L1i cache. It also avoids many unnecessary and redundant prefetch candidates to be inserted into the prefetch queues, which helps useful prefetches to be served faster.

C. wrong_SNL Prefetcher

SN4L may filter some useful prefetches due to complex control flow changes. Consider the following stream of demanded blocks:

$$(...,\ A,\ A{+}1,\ ...,\ A,\ B,\ ...)_n$$

The second time that SN4L is triggered on block A, it prefetches block $A+1$ as it is already demanded. But a discontinuity in block A changes the control flow to block B that makes $A+1$'s prefetching useless. As a result, the next time that block A is accessed, SN4L does not prefetch block $A+1$ as its state is prefetched. But the large number of the blocks that are demanded or prefetched from the last time that $A+1$ is prefetched evicts this block. However, at this time, block $A+1$ is demanded by the processor, and SN4L has filtered a useful prefetch. The *wrong_SNL* prefetcher is designed to help SN4L to detect such complex cases and prefetch the blocks.

The first goal of the wrong_SNL prefetcher is to detect when SN4L has filtered a useful prefetch candidate. To meet this goal, whenever a cache miss occurs, RLU is looked up to find out if SN4L has sent a prefetch for the missing block or not. This is necessary because SN4L may prefetch the missing block, but the cache miss has occurred because of prefetcher's insufficient timeliness. If the missing block misses in RLU, it means that SN4L has wrongly filtered that block.

Wrong_SNL devotes a counter for each block that increments whenever such a case happens. Accordingly, wrong_SNL prefetches a block if the counter exceeds a predefined threshold regardless of SN4L's decision. Because SN4L may occasionally make wrong predictions, a policy that just increments the counter may convert wrong_SNL prefetcher to the N4L prefetcher that prefetches almost all of the blocks. To eliminate this scenario, wrong_SNL needs to have a policy to decrement the counters. Consequently, wrong_SNL periodically decrements the counter of all entries. Based on this policy, a block shows that it needs wrong_SNL's support if new mispredictions by SN4L increments its counter to reach the desired threshold. Otherwise, the counter decrements periodically and SN4L will be the actual prefetcher that makes a decision on prefetching a block.

Wrong_SNL prefetcher uses a table to hold the counters. Similar to SeqTable, this table is direct-mapped and tagless. Each record in this table is a three-bit counter. Whenever a counter reaches three, wrong_SNL prefetches for the block. Moreover, wrong_SNL decrements the counter of all entries every 50 K cycles. Wrong_SNL uses a 64 K-entry table that imposes 24 KB storage overhead.

D. Dis Prefetcher

Having a powerful sequential prefetcher, the remaining misses are discontinuities that occur as a result of branch instructions' execution. While discontinuity prefetcher is already proposed to cover such misses [11], in the previous section, we discussed the shortcomings of its design. In this section, we introduce *Dis* prefetcher that addresses those shortcomings.

First, we show how the problem of having multiple targets can be solved. When a source address has multiple targets, devoting a single target for it loses a significant potential and it may even be harmful because of sending useless prefetches. To clarify this, consider the following sequence of demanded blocks:

$$(...,\ A,\ B,\ ...,\ A,\ C,\ ...)_n$$

Suppose that blocks B and C are always a miss, and as a consequence, DP should update its table. It means that DP repetitively updates the record corresponding to the source block A in its table. When DP finds out B is missed, it updates the table with record $(A,\ B)$ where A is the source, and B is the target. Similarly, when DP finds out C is a miss, it will update the corresponding record to $(A,\ C)$. As such, DP jumps between these two records. Moreover, whenever block A's successor is B, DP holds the record $(A,\ C)$ and incorrectly prefetches C. Similarly, it prefetches B whenever A's succeeding block is C. Such cases show how DP cannot cover the misses and even may make the situation worse because of useless prefetches.

To eliminate these cases, Dis prefetcher exploits the observation that when a source address has multiple discontinuities, the sequence of their appearance can be modeled like this:

$$(..., A, D_1, ..., A, D_2, ..., A, D_i, ..., A, D_k, ...)_n$$

As a result, by knowing the last occurred discontinuity, the next one can be predicted and prefetched. To take advantage of this observation, Dis prefetcher devotes a circular history for each discontinuity instead of a single target field. Whenever a discontinuity occurs, the target address is placed at the tail of the circular history. For prefetching, every time discontinuity prefetcher consults its table; the circular history finds the last target address that is inserted into it, then, it finds the second last insertion of that target, and returns its successor in the circular history as the prediction. Our results show that a 4 entry circular history is sufficient to exploit the available potential.

Devoting a circular history for all observed discontinuities may impose unnecessary storage overhead because a considerable fraction of source addresses have a single target. To better manage the storage budget, Dis prefetcher uses two distinct tables, one for source blocks that have a single target, the other one for those having multiple targets. First, discontinuities are inserted into the first table. Upon an update, if the new target is different from the already recorded target, the discontinuity is moved to the table having a circular history.

To further decrease the storage cost, Dis prefetcher exploits the observation that targets have similar high-order bits with their source address. As a result, instead of storing the complete address of the target, Dis prefetcher stores the low-order bits of the target address. To construct the target address, Dis prefetcher appends the recorded low-order bits of the target to the high-order bits of the source address. Note that source address is available to perform this operation since it is actually the address that is used to lookup the discontinuity table. Accordingly, Dis prefetcher ignores discontinuities that the high-order bits of the source are different from those of the target. By empirical analysis, we find that storing 21 low-order bits of the targets is sufficient to obtain the speedup that is offered when the target address is stored completely.

Moreover, unlike DP, Dis prefetcher uses partial tags for the records stored in its tables. Providing the partial tag for the records helps Dis prefetcher to detect conflicts to eliminate issuing useless prefetches. We find that an 8-bit partial tag is sufficient to distinguish the discontinuity tables' hits from misses.

To summarize, our Dis prefetcher uses two tables, one for source addresses that have a single discontinuity (DisTable_single) and the other one for those sources that have multiple discontinuities (DisTable_multiple). Both tables use an 8-bit partial tag and have a 4-way set-associative structure. These tables use the Least Recently Used (LRU) replacement policy. As a result, we have used a 4 entry queue to hold the LRU order of the entries in each set that requires 8 bits. Moreover, these tables store 21 low-order bits of the target addresses. DisTable_single and DisTable_multiple have 12 K and 4 K entries and impose 46.5 and 48 KB storage overhead, respectively.

E. Triggering Policy and Proactive SN4L+wrong_SNL+Dis Prefetching

Our prefetchers are triggered by *Tagged* policy that prefetching starts when a demand request is missed or already prefetched. After the L1i receives a demand request for a prefetched block, the block does not remain in the prefetched state, and it will be considered a demand. This demand request triggers the sequential prefetchers to find prefetch candidates for the next four subsequent blocks. If they decide to prefetch a block, i.e., the prefetch candidate is not found in RLU, Dis prefetcher is triggered to find a discontinuity prefetch candidate for that block. With this approach, the proposed prefetchers can cover the sequential region and up to one discontinuity ahead of the current demanded block. However, as frequent discontinuities may slow down prefetchers' progress and result in insufficient timeliness, we enhance the triggering policy to enable the prefetchers to go multiple sequential and discontinuity regions ahead, as needed. Such a proactive prefetching mechanism is triggered as follows. When SN4L or wrong_SNL prefetch a block, Dis is triggered for those blocks, and whenever Dis prefetches a block, all SN4L, wrong_SNL, and Dis prefetchers are triggered for that block. In other words, the proposed prefetchers are triggered to prefetch sequential regions of discontinuities (*SeqOnDis*), discontinuity of a discontinuity (*DisOnDis*), and discontinuity of the sequential regions (*DisOnSeq*). However, this policy can make progress without any limit. Consequently, we assign a counter to each issued prefetch, and when a prefetch request triggers another prefetch, we increment the counter. We terminate the chain of prefetches when the counter reaches a predefined threshold. We find seven to be an appropriate threshold in the context of the given traces.

Once the prefetch candidates are determined, they are pushed into the *RLUQueue* to be looked up in RLU. Those prefetch candidates that are in RLU are discarded. Those prefetch candidates that miss in RLU are looked up in the L1i cache, and a prefetch request is sent for those that miss in the L1i. We use a 32-entry SeqQueue, DisQueue, and RLUQueue. Moreover, RLU holds 8 entries. These structures require 0.81 KB altogether.

F. Total Storage Cost

Our proposed prefetcher uses a 64 K entry SeqTable and a 64 K entry wrong_SNL_table that require 8 and 24 KB, respectively. DisTable_single and DisTable_multiple need 46.5 and 48 KB in the given configuration. SeqQueue, DisQueue, RLUQueue, and RLU impose 0.81 KB altogether to our design. Summing up the storage requirements of these components together, our proposed prefetcher has used 127.31 KB that meets the competition rules.

G. Dis Prefetching and Predecoding

In recent years, some proposals have leveraged the instruction predecoding to eliminate frequent BTB misses, and hence, obtain better performance [5], [7], [8]. Such a notion can be used for our Dis prefetcher to reduce its storage requirements significantly. As discontinuities are the consequence of branch instructions' execution, instead of recording the target block, Dis prefetcher can record the offset of the branch instruction in a cache block that has caused the discontinuity. Then Dis prefetcher can extract the target block by predecoding the corresponding instruction. Assuming 64-byte block size and a fixed-length instruction set architecture that instructions are 32-bit long (like SPARC v9 ISA), 4 bits are required to determine the branch instruction inside a cache block. This is significantly lower than the current implementation that records 21 low-order bits of the target block to reconstruct it using the source address. Moreover, this design can be applied to variable-length ISAs by recording the byte offset of the branch instruction in a cache block, which needs 6 bits. However, we do not use such optimizations because the competition does not provide the required information in its given interface.

III. Evaluation

A. Methodology

We evaluate our proposal in the context of the simulation framework provided with IPC-1 [2]. We use ChampSim [1] with all default configurations and follow the evaluation methodology of the

championship and run simulations for all 50 provided traces. Each trace is executed for 50 million instructions to warm-up the caches, branch predictors, and the required metadata for the prefetchers. The subsequent 50 million instructions are used to collect the Instruction Per Clock (IPC) metric. We compare our proposal, SN4L+wrong_SNL+Dis prefetcher with DP [11], RDIP [6], Shotgun [7], and PIF [3] prefetchers. The figures show the obtained results for ten selected benchmarks, the average of those ten, and the average of all fifty traces.

B. Results

1) Performance Breakdown: Figure 1 shows how different proposed prefetchers contribute to the final speedup offered by SN4L+wrong_SNL+Dis prefetcher. NL (Default) is an implementation of the next-line prefetcher that is provided in ChampSim [1]. Dis (Single) refers to a Dis prefetcher that just uses DisTable_single. On the other hand, Dis (Multiple) benefits from both tables. Figure 1 shows how SN4L offers a considerably better speedup as compared to NL and N4L prefetchers. Moreover, a substantial speedup is obtained by SN4L+Dis (Single) prefetcher that contributes to a smaller fraction of the storage cost required by all prefetchers that are used in SN4L+wrong_SNL+Dis prefetcher. Moreover, this figure shows how augmenting the other prefetchers to SN4L+Dis (Single) helps to cover the remaining potential to offer 25% speedup as compared to the baseline design.

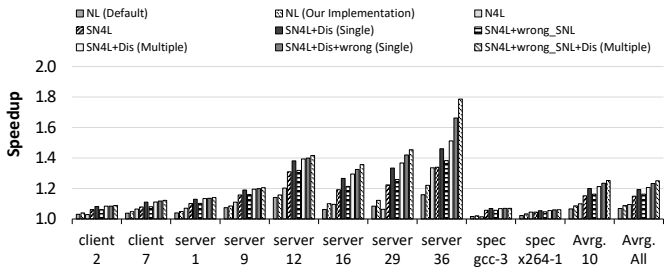

Fig. 1. Performance breakdown of our proposal.

2) Comparison with competitors: Figure 2 shows the obtained speedup when different storage budgets are provided for competing prefetchers. When we utilize all the provided 128 KB storage budget, SN4L+wrong_SNL+Dis prefetcher offers 25% speedup, which is the highest among the competitors. Moreover, PIF and Shotgun are the other proposals that can provide the same-level of speedup. On the other hand, DP and RDIP lag considerably behind the others. When we limit the storage cost, we see that our proposal still has the upper hand over the other prefetchers. Our proposal offers 15 and 18% speedup when only 8 and 16 KB storage budget is provided to it. This is over 3% higher than what is offered by the closest competitor, Shotgun, that provides 12 and 15% speedup. For Shotgun, we have assumed that the baseline design uses a 2 K entry BTB that requires 23.7 KB storage [7]. As a result, the storage budget is used to enlarge Shotgun's BTBs. For example, in our implementation of Shotgun whose storage budget is 8 KB, BTBs have 40 KB storage[1]. In general, RDIP offers poor performance because of two reasons. First, each prefetching record in RDIP requires more than 20 bytes. As a result, RDIP's table can hold a fewer number of records in its table in a given budget. Moreover, RDIP creates a considerably larger number of distinct prefetching records. In consequence, we see that even providing 128 KB storage budget is not sufficient for RDIP to reach the level of performance that is offered by the competing prefetchers when they use a similar storage budget. We also evaluated an implementation of RDIP with an infinite storage budget and observed that this implementation offers 23% speedup, which is close to that of the competitors. Finally, we see the obtained speedup by DP prefetcher. Comparing N4L with DP-8 KB, it can be concluded that DP-8 KB offers lower speedup because of a large number of useless prefetches as the consequence of the conflicts in the discontinuity table that cannot be detected. In DP-128 KB, DP can hold a larger number of records, and hence, conflicts are decreased. In consequence, DP-128 KB offers 13% speedup, which is an improvement compared to the N4L prefetcher. However, DP-128 KB has a significant gap with our proposal. It clearly shows how our modifications to the conventional sequential and discontinuity prefetchers resulted in a significant improvement.

[1] In the authors-proposed configuration of Shotgun, it imposes 6 KB storage cost for its prefetching buffers and the changes that are made to the BTB. However, we have ignored this storage cost to reflect the impact of investing the storage budget on the actual metadata that a prefetcher uses.

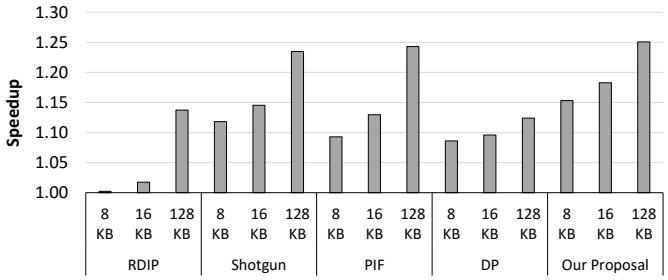

Fig. 2. Speedup offered by competing prefetchers with different storage budgets.

REFERENCES

[1] "ChampSim," https://github.com/ChampSim/, 2020.
[2] "IPC-1," https://research.ece.ncsu.edu/ipc/, 2020.
[3] 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.
[4] 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.
[5] C. Kaynak, B. Grot, and B. Falsafi, "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.
[6] 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.
[7] 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.
[8] 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.
[9] 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.
[10] A. J. Smith, "Sequential Program Prefetching in Memory Hierarchies," *Computer*, vol. 11, no. 12, pp. 7–21, Dec. 1978.
[11] L. Spracklen, Y. Chou, and S. G. Abraham, "Effective Instruction Prefetching in Chip Multiprocessors for Modern Commercial Applications," in *Proceedings of the 11th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2005, pp. 225–236.