2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)

Divide and Conquer Frontend Bottleneck

Ali Ansari[‡], 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*—The frontend stalls caused by instruction and BTB misses are a significant source of performance degradation in server processors. Prefetchers are commonly employed to mitigate frontend bottleneck. However, next-line prefetchers, which are available in server processors, are incapable of eliminating a considerable number of L1 instruction misses. Temporal instruction prefetchers, on the other hand, effectively remove most of the instruction and BTB misses but impose significant area overhead.**

Recently, an old idea of using BTB-directed instruction prefetching is revived to address the limitations of temporal instruction prefetchers. While this approach leads to prefetchers with low area overhead, it requires significant changes to the frontend of a processor. Moreover, as this approach relies on the BTB content for prefetching, BTB misses stall the prefetcher, and likely lead to costly instruction misses. Especially as instruction misses are usually more expensive than BTB misses, the dependence of instruction prefetching to the BTB content is harmful to workloads with very large instruction footprints. Moreover, BTB-directed instruction prefetchers, as proposed in prior work, cannot be applied to variable-length ISAs.

In this work, we showcase the harmful effects of making instruction prefetchers depend on the BTB content. Moreover, we divide the frontend bottleneck into three categories and use a divide-and-conquer approach to propose simple and effective solutions for each one. Sequential misses can be covered by an accurate and timely sequential prefetcher named *SN4L*, a lightweight discontinuity prefetcher named *Dis* eliminates discontinuity misses, and the BTB misses are reduced by pre-decoding the prefetched blocks. We also discuss how our proposal can be used for variable-length ISAs with low storage overhead. Our proposal, *SN4L+Dis+BTB*, imposes the same area overhead as the state-of-the-art BTB-directed prefetcher, and at the same time, outperforms it by 5% on average and up to 16%.

***Index Terms*—Frontend bottleneck, instruction and BTB prefetching, divide and conquer**

I. Introduction

Server workloads significantly suffer from the frontend bottleneck due to their massive instruction footprints [1]–[6]. The instruction footprints of server workloads are responsible for a substantial number of instruction-cache, and branch-target-buffer (BTB) misses. Such misses considerably degrade the performance of server processors [1], [4], [7].

Instruction prefetching is a widely-used approach to address the frontend bottleneck of server processors. Existing commercial processors benefit from a sequential prefetcher that sends prefetch requests for a number of subsequent blocks upon activation [8]. Unfortunately, sequential prefetchers leave a significant fraction of misses uncovered, exposing considerable performance degradation to server workloads. Due to limitations of sequential prefetchers, researchers proposed a number of advanced prefetchers to address the frontend bottleneck [9]–[20].

Discontinuity prefetcher is a simple approach to assist sequential prefetchers [17]. While a sequential prefetcher covers sequential misses, a discontinuity prefetcher attempts to eliminate the remaining misses. Such a prefetcher records the discontinuities of the program control flow for which the L1i cache encounters a miss. However, discontinuity prefetchers have some shortcomings (e.g., limited lookahead). These shortcomings motivated researchers to propose sophisticated and complex techniques to cover the instruction cache misses [14].

Many of the advanced prefetchers are based on temporal prefetching [14]–[16], [21]. In such a prefetcher, the sequence of past misses [14] or accesses [15] are recorded and replayed to predict and prefetch future instruction misses/accesses. While such prefetchers are effective at eliminating most of the misses, they impose significant area overhead, as the sequence of past misses or accesses needs to be recorded, and server workloads have a large number of instructions.

To address the drawback of temporal instruction prefetchers, Kumar et al. revived *BTB-directed (also called fetch-directed) instruction prefetching* and extended it to prefetch for the BTB in addition to the instruction cache [19]. The proposed prefetcher, which is called Boomerang, uses BTB to go ahead in the instruction stream to determine instruction and BTB misses. Then Boomerang sends prefetch requests for the missing instruction blocks and branches in the cache and BTB, respectively.

Despite BTB prefilling in Boomerang, BTB misses are still a bottleneck for this prefetcher. As BTB misses prevent such schemes from going ahead of the current instruction stream, and BTB misses are resolved using pre-decoding the instruction blocks, such schemes are not useful for workloads with very large instruction footprints [20], where BTB misses are frequent. To lower the effect of BTB misses, recently, Kumar et al. proposed a new BTB organization within the framework of BTB-directed instruction prefetching. The new BTB organization is used in a prefetcher called Shotgun [20]. While Shotgun lowers the number of BTB misses, and as such, offers higher performance improvement on workloads with large instruction footprints, it still suffers from BTB misses, and hence, does not provide the full potential on these workloads. Moreover, prefetchers that rely on BTB-directed instruction prefetching require significant changes to the frontend of a processor. Last but not least, prior instruction and BTB prefetchers [16], [19], [20] are proposed in the context of fixed-length ISAs. Naively extending them to support variable-length ISAs imposes significant overhead.

In this paper, we propose a low-cost instruction and BTB prefetcher that requires minimal changes to the frontend of a processor and offers a level of performance that exceeds

978-1-7281-4661-4/20/$31.00 ©2020 IEEE
DOI 10.1109/ISCA45697.2020.00017

that of the state-of-the-art prefetchers on workloads with very large instruction footprints. Moreover, we discuss how our proposal can be used for a variable-length ISA with minimal storage overhead. To achieve this goal, we use a divide-and-conquer approach: we divide misses into three categories, sequential, non-sequential (discontinuity), and BTB misses, and offer practical prefetchers for each category by taking advantage of the following contributions.

1) We observe that for sequential prefetching, there is a trade-off between timeliness and accuracy. On the one hand, next-line (NL) prefetchers fall short of efficiency because of poor timeliness. On the other hand, more aggressive sequential prefetchers (e.g., next-four-line (N4L) prefetcher) are inaccurate and produce a large number of useless prefetches. To address this fundamental problem, we augment an N4L prefetcher with a simple-yet-accurate predictor to identify which ones of the subsequent four blocks are useful and will be accessed by the processor. Only those blocks that are predicted to be useful will be prefetched. As a result, our sequential prefetcher is both timely and accurate.
2) Eliminating the sequential misses, we assess a discontinuity prefetcher to cover the remaining misses. We observe that the conventional discontinuity prefetcher cannot meet this goal because of three main shortcomings: (1) high storage cost, (2) useless prefetches, and (3) the limited lookahead. We propose a variant of the discontinuity prefetcher to address these shortcomings. For the storage cost, we leverage the fact that discontinuities are the result of branch instruction's execution. In consequence, instead of recording the target address, we record the instruction offset of the branch instruction in a cache block and extract the target through instruction predecoding. Moreover, useless prefetches are the consequence of using a tagless table in the conventional discontinuity prefetcher to mitigate its high storage cost. We solve this problem using a partial-tagged table with low storage overhead. Finally, we introduce a proactive sequential and discontinuity prefetcher to provide sufficient lookahead.
3) Having a powerful and low-cost instruction prefetcher, we adopt a Confluence-like BTB prefetcher [16] to cover BTB misses. Every time a block of instruction is accessed, its instructions are decoded to determine branch instructions. Branch instructions are sent to the BTB prefetch buffer to eliminate future BTB misses. Unlike prior work [16], [19], [20] that only considered fixed-length ISAs, we show how our proposal can be used in the context of variable-length ISAs with a low area overhead.

II. Background

In this section, we review some of the main trends in the instruction and BTB prefetching and their limitations.

A. Temporal Prefetchers

The primary instruction prefetching technique for servers in the past ten years is temporal prefetching. With temporal prefetching, the sequence of prior cache accesses [15] or misses [14] is recorded and replayed to eliminate cache misses. The access-based temporal instruction prefetcher [15] is strong and eliminates most of the cache misses but imposes significant storage overhead (e.g., 200 KB per core [15]).

The main focus of research in temporal prefetching was the reduction of the storage overhead. One proposal to reduce the storage overhead is to share the metadata (i.e., the sequence of past cache accesses) among all the cores in a multi-core processor and virtualize it in the last-level cache (LLC) [21]. While this proposal significantly reduces the overall storage overhead for homogeneous workloads, it does not work for cases where several workloads are simultaneously running on a multi-core processor.

Another significant improvement in this area is Confluence [16] that extends the instruction prefetcher to prefill for the BTB as well. The crucial observation was that the prefetched blocks have all the necessary information to prefill the BTB and avoid BTB misses. As blocks of instructions come to the cache, their instructions are decoded, and the identified branch instructions are inserted into the BTB. This way, the instruction prefetcher is effectively used to prefetch for the BTB, eliminating the storage overhead of a BTB prefetcher.

B. BTB-Directed Prefetchers

Despite all the efforts, temporal instruction prefetchers still impose significant overhead. BTB-directed prefetchers require a significantly lower storage cost as compared to temporal prefetchers by leveraging the branch target buffer that is already available in a conventional processor [10], [19], [20]. The main idea is to go ahead in the instruction stream by identifying the branch instructions using BTB and taking advantage of the branch predictor to determine the targets of the branch instructions. Those blocks in the instruction stream that are not in the cache are brought into the cache (i.e., prefetched) before the processor demands for them. The need for a near-ideal BTB to correctly go ahead of the fetch stream is the main drawback of BTB-directed prefetchers.

Recently Kumar et al. [19] revived the old idea of BTB-directed prefetching by extending it to prefetch for the BTB in addition to the cache in a prefetcher named Boomerang. They used a basic-block oriented BTB that enables Boomerang to detect BTB misses. By identifying BTB misses, Boomerang fetches or prefetches the instruction block that contains the missing BTB entries. Then it uses an instruction pre-decoder to pre-decode the instructions to extract the missing BTB entries to fill in the BTB. Having the required BTB entries, Boomerang can continue its progress to discover the instruction blocks ahead of the fetch stream.

While this idea has low area overhead and works well for server workloads with modestly-sized instruction footprints, it falls short of efficiency for workloads with large instruction footprints where BTB misses are frequent [20]. To address this limitation, Kumar et al., in their latest work [20], suggested changes to the BTB of a processor to make BTB misses less likely, and hence, the resulting BTB-directed prefetcher, named Shotgun, suitable for a broader class of workloads (i.e., workloads with larger instruction footprints). They split a BTB into three parts: (1) C-BTB for conditional branches, (2) U-BTB for unconditional branches, and (3) RIB for return instructions. The main idea is to use most of the BTB storage for unconditional branches while using BTB prefilling for conditional branches.

Moreover, upon detecting an unconditional or a return branch instruction, for prefetching, Shotgun relies on bit vectors of useful blocks around the branch instruction or its target instead of using the branch predictor, as in the earlier work. These changes enable Shotgun to be more effective on workloads with large instruction footprints when it uses the same BTB size as Boomerang.

III. Why not Shotgun?

To overcome Boomerang's limitations, Shotgun dedicates a large part of BTB to the unconditional branches (i.e., U-BTB). Moreover, for each unconditional branch, it stores the *return* and *call* footprints to have the working set around the branch instruction and its target. Such a BTB can hold a larger address space as compared to the original BTB. Moreover, when an unconditional or return branch is touched, the prefetch candidates are quickly extracted from the footprints independent of the BTB content.

As Shotgun dedicates a large part of the BTB to U-BTB, to keep the BTB size small, it only dedicates a small part of the BTB to conditional branches (i.e., C-BTB). Instead, Shotgun attempts to aggressively prefill C-BTB by decoding the instruction blocks.

Shotgun requires a meager U-BTB miss ratio to perform well. We observed that for workloads with enormous instruction footprints in which U-BTB is not sufficient to hold the unconditional branches, Shotgun is ineffective. This ineffectiveness is due to the following reasons. While the BTB prefilling mechanisms can resolve some missing U-BTB entries, their footprints cannot be prefilled because they are constructed using the retired instruction streams. Not having the footprints, Shotgun cannot benefit from its instruction prefetcher as well as the proactive prefilling to fill in C-BTB. Consequently, instruction and C-BTB misses are likely to happen.

In the case of a footprint miss, Shotgun relies solely on reactive prefilling: when a C-BTB miss happens, it performs a cache lookup, sends a prefetch request in the case of a cache miss, waits for the reply, decodes the block, fills the C-BTB, feeds it into the *Fetch Target Queue* (FTQ)[1], and makes progress till the next C-BTB miss. Even though Boomerang also uses a similar process, unlike Shotgun, Boomerang can quickly make progress because conditional branches are kept in its large BTB. For Shotgun, however, frequent C-BTB misses, which are caused by footprint misses, make the core to have forward progress one block of instructions at a time as in the baseline with no prefetcher.

Because of the mentioned problems, in the case of U-BTB misses, FTQ cannot be filled by instructions far ahead in the fetch stream. The fetch engine consumes the instructions that are already inserted into the FTQ, and FTQ quickly becomes empty, which stalls the core and results in performance loss (note that in the case of a BTB miss, Shotgun stops inserting into the FTQ).

Figure 1 shows the footprint miss ratio in the U-BTB. We report footprint miss ratio instead of U-BTB miss ratio because BTB prefilling may prefill some U-BTB entries[2]. Still, their footprints must be constructed using the retired instruction streams, and as such, BTB prefilling is unable to fill the footprints. As Figure 1 shows, footprint misses are frequent and footprint miss ratio varies from 4 to 31%.

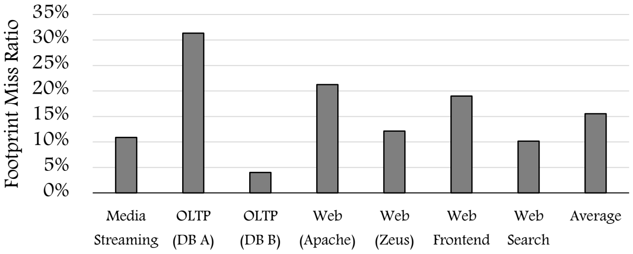

Fig. 1. Footprint miss ratio in Shotgun [20].

Due to the high footprint miss ratio, we expect the discussed shortcomings to influence Shotgun's effectiveness significantly. To give an insight into the adverse effects, we report the fraction of cycles that cores are stalled due to empty FTQ in Table I. Note that empty FTQ is a consequence of slow progress on feeding the basic blocks to the FTQ.

TABLE I
Empty-FTQ stall cycles in Shotgun [20].

Workload	Fraction
Media Streaming	13.13%
OLTP (DB A)	18.87%
OLTP (DB B)	1.64%
Web (Apache)	14.58%
Web (Zeus)	14.10%
Web Frontend	8.47%
Web Search	8.16%

IV. Motivation

L1 instruction (L1i) cache misses can be classified into sequential and discontinuity misses. A sequential miss is a cache miss that is spatially right after the last accessed block, while a discontinuity miss is the consequence of a *branch* execution. We propose two prefetchers to prefetch these two classes of misses effectively.

First, we start with sequential misses, which are the majority of L1i cache misses. As shown in Figure 2, 65 to 80% of L1i cache misses are next to the last accessed block. Consequently, a sequential prefetcher has the potential to cover a significant fraction of L1i misses.

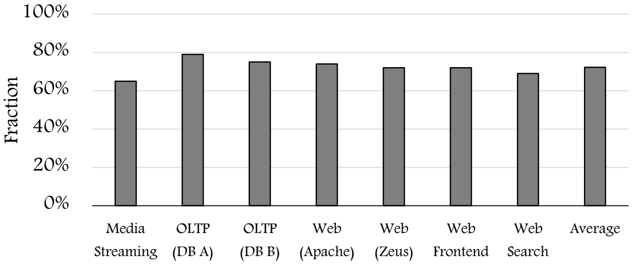

Fig. 2. Fraction of sequential cache misses.

[1] FTQ is a long queue of basic-blocks, which is used to fill the gap between the branch prediction unit and the instruction cache.

[2] Entries corresponding to branch instructions whose target is encoded in the instruction itself like *Call*.

Spurred by the results, a sequential prefetcher can be used to eliminate sequential misses. Sequential prefetchers have different variants like Next-Line (NL), Next-X-Line (NXL), NLmiss, and NLtagged [22]. The simplest one, the NL prefetcher, upon access to the cache, looks up the cache for the next cache block and prefetches it if it is not in the cache. Unfortunately, as we show, the NL prefetcher leaves a considerable fraction of sequential misses uncovered.

Figure 3 shows the *sequential* miss coverage of an NL prefetcher over a baseline with no prefetcher. Based on detailed cycle-accurate simulations, the average sequential miss coverage is 63% (i.e., 37% of the misses are not covered). Note that this figure only considers sequential misses and differs from common miss coverage reports in which discontinuity misses are also considered.

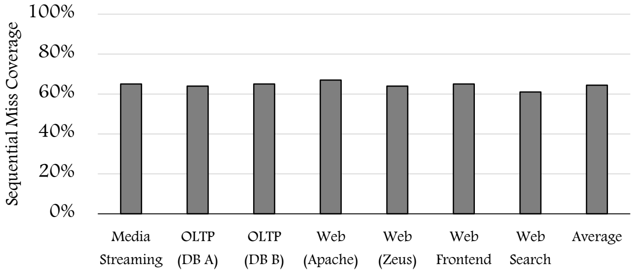

Fig. 3. NL sequential miss coverage.

A timely NL prefetcher must cover all sequential misses. Consequently, NL prefetcher's poor timeliness is the source of the remaining uncovered sequential misses. The general approach to improve the timeliness of a sequential prefetcher is to increase its prefetching depth by prefetching the next X blocks, which is called Next-X-Line or simply NXL prefetcher. To measure the timeliness, we use the covered memory access latency (CMAL): the fraction of cycles needed to fetch the cache block from lower levels of the memory hierarchy that is covered by the prefetcher. Figure 4 shows the average timeliness of NL, N2L, N4L, and N8L. Results show that NL's CMAL is 65%, confirming its poor timeliness. As expected, increasing the prefetch depth improves the timeliness: CMAL for N2L, N4L, and N8L is 80%, 88%, and 85%, respectively. Surprisingly, N8L offers lower CMAL as compared to N4L. Investigating the reason, we find that the excessive N8L's useless prefetches increase the network latency, and lead to weaker CMAL for the blocks that are prefetched with small depths. Based on this insight, we analyze the impact of useless prefetches in the context of sequential prefetchers.

One of the significant limitations of a sequential prefetcher is that it issues many useless prefetches, which leads to essential side effects like cache pollution, external bandwidth utilization, and excessive network traffic [6], [23], [24]. Server applications are sophisticated software with many conditional branches, subroutine, and procedure calls, exception handling cases, and error detection and debugging pieces of code, which are rarely executed. These rarely-executed pieces of code make a sequential prefetcher issue many incorrect prefetches. As an example, consider the piece of code in Algorithm 1.

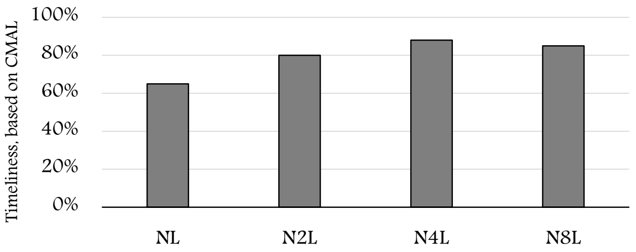

Fig. 4. Covered Memory Access Latency (CMAL) for different sequential prefetchers.

Algorithm 1 A sample procedure

```
procedure PROCEDURE
    some calculations                          ▷ Block $A$
    if some conditions then                    ▷ Block $A+1$
        some calculations
    else                                       ▷ Block $A+2$
        some calculations
    end if
    try                                        ▷ Block $A+3$
        some calculations
    catch                                      ▷ Block $A+4$
        handle some exceptions
    some calculations                          ▷ Block $A+5$
    return
end procedure
```

In this example, Block A will not be prefetched by an NL prefetcher because its previous line, *A-1*, belongs to another procedure and is not accessed. On access to Block A, an NL prefetcher correctly prefetches *A+1* whose *if condition* must be evaluated. If the condition causes the *if* body to be executed, the Block *A+2* will not be executed (i.e., the *else* part), which results in a useless prefetch. After the *if* statement, the *try* block will be executed, which is placed in Block *A+3* and is not prefetched because the *else* block, *A+2*, is not accessed. Meanwhile, the rarely-executed exception handling *catch* block is prefetched, which is yet-another useless prefetch. The same case holds when the NL prefetcher prefetches the out-of-the-procedure block, *A+6*. Although *A+3* and *A+5* are accessed from branch instructions located at the end of the *if* and *try* blocks, and are not sequential misses, they can be covered by a next-2-line prefetcher.

To study the significance of useless prefetches of a sequential prefetcher in server workloads, we measure the average clock cycles needed to access the last-level cache (LLC) and the L1i external bandwidth usage of NXL prefetchers normalized to the baseline with no prefetcher. External bandwidth usage refers to the number of fetch or prefetch requests sent from L1i to the lower levels of the memory hierarchy. In this study, we use a 64-entry prefetch buffer along with the L1i to immune it from cache pollution. Based on the results shown in Figure 5, as compared to the baseline, the average LLC latency and the external bandwidth usage of the N8L prefetcher are increased by 28% and 7.2×, respectively, because of sending useless prefetches. As a result, while increasing the prefetch depth addresses the timeliness, the useless prefetches bring essential side effects that significantly limit the effectiveness of a sequential prefetcher.

As an N4L prefetcher offers sufficiently good timeliness, we

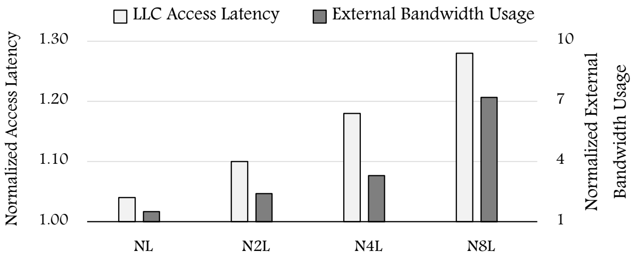

Fig. 5. Side effects of useless prefetches on LLC access latency and L1i external bandwidth usage.

choose it as our starting sequential prefetcher and attempt to eliminate its useless prefetches. For this purpose, we measure the predictability of accesses to the subsequent blocks. We observe that for each block of instructions, the four subsequent blocks' access pattern is stable and can be learned. For each block, from its insertion to the cache until its eviction, we record which of the four subsequent blocks are accessed, and compare this pattern to the last recorded one to find out the predictability. Figure 6 shows that the accuracy of this prediction is 92%, on average. Based on this observation, we use a simple predictor to learn the access pattern and use it to eliminate the useless prefetches of an N4L prefetcher. This process is so effective that the number of useless prefetches of the N4L prefetcher with such a predictor becomes significantly smaller than that of an NL prefetcher.

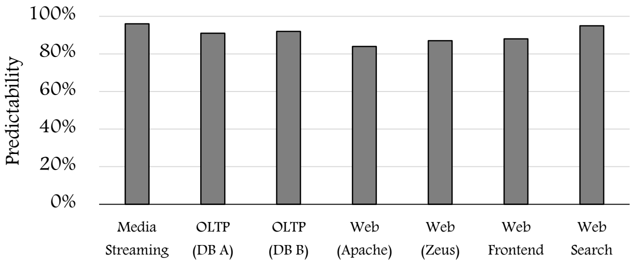

Fig. 6. Predictability of the access pattern of the four subsequent cache blocks of a cache access.

Sequential misses being covered, the remaining misses are non-sequential ones caused by branch instructions. We use the term discontinuity for these misses. We observe that in most cases, only a single branch instruction within a cache block is responsible for the discontinuity. To show this, for each block, we compare two consecutive branch instructions that caused the discontinuity. Figure 7 shows the fraction of cases where an identical instruction causes a discontinuity. As indicated, the number ranges from 78% in Web (Apache) to 83% in OLTP (DB A), with an average of 80%. Based on this observation, we use a predictor to learn which single branch within a block is responsible for the discontinuity. Knowing the branch, we easily prefetch the target of the branch to avoid misses, with a small area cost.

While BTB prefetching using instruction-block pre-decoding is already proposed by prior work [16] and used in the followup work [19], [20], its implementation on a variable-length instruction set architecture (VL-ISA) has never been discussed. On fixed-length ISA, the instruction boundaries are known, and the pre-decoder can easily find the instructions and check their opcodes to determine branches and extract their targets. Nevertheless, detecting instruction boundaries is a challenge for VL-ISA. We want to offer an approach with minimal overhead to make BTB prefetching applicable to VL-ISA.

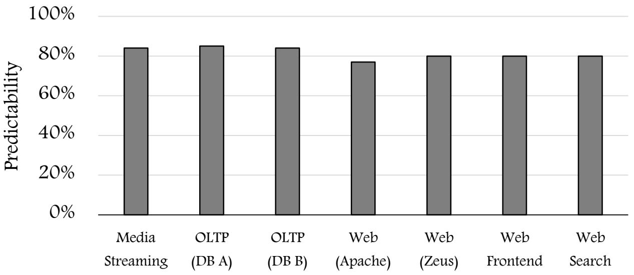

Fig. 7. Predictability of the branch instruction responsible for the discontinuity.

To avoid the complexity of detecting the instructions and determining their boundaries inside a cache block, we need to know where the branch instructions in the block are located. Moreover, unlike a fixed-length ISA where instructions within a cache block can be pre-decoded in parallel, in a VL-ISA, the progress is instruction by instruction and slows down the prefetcher and can offset the benefits.

For this goal, we need to make some critical decisions. First, we need to decide how a branch instruction within a cache block should be identified. A simple solution is to keep the byte offset of the starting byte of the branch instruction, which requires 6 bits in a 64-byte block. Second, we need to decide how many branches should be identified for a cache block. Figure 8 shows that by only storing four branches per cache block, almost all branches can be identified. As each byte-offset needs 6 bits, this imposes 3 bytes of storage per cache block. We refer to the combination of all branch byte-offsets of a cache block as a branch footprint (BF).

Naively storing the BF of all instruction blocks imposes a high storage cost. We note that it is not needed to store all BFs near L1i. They can be kept in the LLC: when a cache block is fetched from the LLC, its corresponding BF is fetched as well. Moreover, we virtualize the BFs to make the extra storage overhead as low as possible. A convenient virtualization approach is to extend the tag of a block in the LLC to store, update, and retrieve its BF. Such a virtualization approach is already used in SHIFT [21] by dedicating some sets to store the virtualized metadata and extending the tag array to store the indexes to the virtualized sets. However, this approach is not a good solution because instruction blocks are a small fraction of all blocks in the LLC: a method that requires even small extra storage to all cache blocks will result in considerable storage overhead.

To reduce the storage overhead, we propose to keep the BFs per cache set and not per cache block. As many of the blocks in a set are not instruction blocks, the storage overhead will be reduced. To show the effectiveness of this insight, Figure 9 shows the fraction of uncovered branch footprints as a function of the number of stored branch footprints per set. Note that a branch footprint has the footprint of four branches of a cache block. Figure 9 shows that by providing space to store two BFs per LLC set, about 2% of the BFs remain uncovered. Increasing the area to accommodate three and four BTs decreases the uncovered ratio to 0.4 and 0.2%, respectively. Consequently, by storing only four BFs per set in the LLC, the BF of almost all instruction blocks are covered.

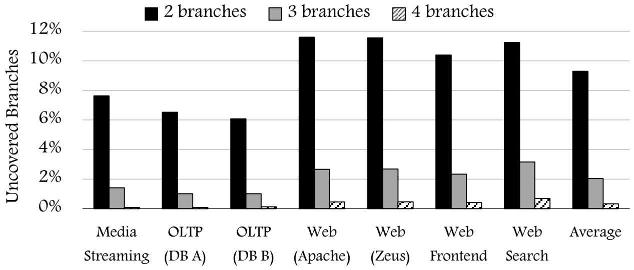

Fig. 8. Fraction of uncovered branches as a function of the number of branches stored in a branch footprint.

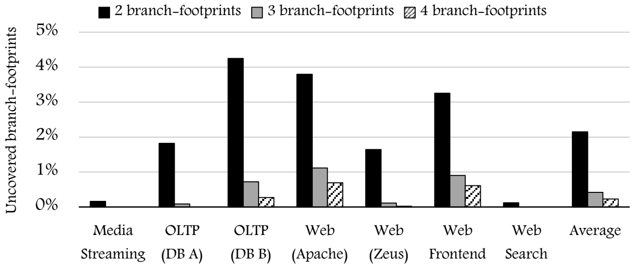

Fig. 9. Fraction of uncovered branch footprints as a function of the number of dedicated branch footprints per LLC set.

V. SN4L+DIS+BTB

SN4L+DIS+BTB prefetcher is a combination of a sequential instruction prefetcher, named SN4L, to cover sequential instruction misses, a discontinuity prefetcher to cover non-sequential instruction misses and a BTB prefetcher, that synergistically work together. This section presents the details of the SN4L+DIS+BTB prefetcher.

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

We describe selective-next-four-line (SN4L) prefetcher, which is based on a next-four-line prefetcher (N4L) that covers most of the sequential misses but attempts to eliminate N4L's useless prefetches.

Upon every access to a cache block, SN4L attempts to prefetch the next four subsequent cache blocks. Unlike an N4L prefetcher that sends prefetch requests for all four subsequent cache blocks that are not in the cache, the SN4L prefetcher only sends prefetch requests for those of the four blocks that are not in the cache and were useful when prefetched the last time. As SN4L is highly accurate, it prefetches directly into the cache and does not need a prefetch buffer.

SN4L needs to store the state of the usefulness of the prefetched cache blocks. Therefore, every block of instructions has a single-bit sequential prefetch status that indicates if the block should be sequentially prefetched or not. These states are stored in a direct-mapped and tagless table named SeqTable. All entries in the table are initialized to 1, which indicates that all blocks should be prefetched the first time. In addition to SeqTable, as in most prefetchers, every block in the cache has a 1-bit prefetch flag. The flag indicates whether the cache block is brought into the cache by the prefetcher or the fetch demand. Finally, we store 4-bit local prefetch status in every block in the cache, as we explain in the following section.

Decreasing SeqTable lookups: As SN4L is triggered on every access to a block, it may impose significant SeqTable lookup overhead. In consequence, for each block in the cache, we allocate 4-bit local prefetch status in the cache to store the prefetch status of its four subsequent blocks. When we bring a block into the cache, the prefetch status of its four subsequent blocks is read from SeqTable and stored in the local prefetch status. Every time we need to read the status of the four subsequent blocks, we use the 4-bit local prefetch status in the cache. This strategy eliminates unnecessary access to SeqTable.

Prefetching: Upon access to a cache block, SN4L checks the 4-bit local prefetch status to determine the status of its four subsequent blocks. For all of the four subsequent cache blocks that are not in the cache and their prefetch status bits indicate that they are useful, a prefetch request will be sent.

Updating the metadata: Every time the core demands for a prefetched block (i.e., a block whose prefetch flag is set), its prefetch status in SeqTable is set to indicate that this block is a useful prefetch. Moreover, upon demand access to a prefetched block, we reset the prefetch flag of the cache block. On the other hand, every time a cache block is evicted from the cache, if the evicted block is prefetched, the corresponding entry in SeqTable is reset to show that the block was not a useful prefetch. Additionally, on a missed access to a block, the SeqTable entry for that block is set.

Metadata table: As SN4L requires 1-bit prefetch status for every block of instructions, the size of SeqTable can be quite large. However, we use a direct-mapped and tagless table to moderate the storage cost. As the size of the table is limited, every entry in the table is shared for many addresses. In SeqTable, if a block is mapped to entry *A*, the four subsequent blocks' prefetch status are stored in the entries *A+1* to *A+4*. Moreover, Block *A+1*'s four subsequent blocks' prefetch status are stored in entries *A+2* to *A+5*. Our experiments show that such a table successfully stores the metadata and offers the performance of dedicated storage for each block while imposing significantly less storage overhead.

B. Discontinuity Prefetcher

The goal of a discontinuity prefetcher is to cover non-sequential misses that occur as a result of the execution of branch instructions. While a discontinuity prefetcher by itself is not a new idea [17] and its organization is straightforward, our goal is to minimize the area overhead of such a prefetcher.

The straightforward implementation of a discontinuity prefetcher, similar to that of prior work [17], is a table that records the miss addresses (along with other metadata) that a next-line prefetcher cannot capture. Such a table needs tens of kilobytes of storage as every row of the table needs to store an address, along with other details.

To minimize the storage requirement, we take advantage of the fact that discontinuities are the consequence of the execution of branch instructions. As a result, if a block of instructions has a branch that causes a discontinuity miss, the offset of the branch in the block is recorded in a table named DisTable. Accordingly, we record a 4-bit offset to distinguish 16 instructions in a block. To make DisTable simple, we organized DisTable as direct-mapped and partially-tagged. In consequence, each row contains a 4-bit partial tag of the cache line and a 4-bit offset of the instruction in the line. We refer to our discontinuity prefetcher as the *Dis* prefetcher. Like SN4L, the Dis prefetcher is accurate and prefetches directly into the cache and does not need a prefetch buffer.

Recording: The last two demanded instructions and their program counters are recorded in individual registers. Initially, the DisTable is empty. Upon every cache miss, the last two demanded instructions are decoded to determine if they are branch instructions[3]. If any of them is a branch instruction, the 4-bit offset of the branch instruction is recorded in the entry of the DisTable associated with the block address.

Replaying: Dis prefetcher is triggered whenever there is a fetch or prefetch request to the cache. If the block is in the cache, we pursue the following procedure immediately. However, for blocks that are not in the cache, we pursue the procedure when the block arrives in the cache. We look up DisTable to check if there is an offset associated with the block address. If we find the offset, the offset of the branch instruction is read, and the corresponding instruction will be decoded. Upon decoding the instruction, if the instruction is not a branch instruction, we do nothing. Otherwise, if the branch's target is decoded in the instruction, we send a prefetch request for the target. Otherwise, we consult BTB to determine the target and send the prefetch request. If the instruction is not found in BTB, no prefetch request will be sent.

Proactive Sequential and Discontinuity Prefetching: While SN4L attempts to offer timely prefetches, frequent discontinuities prohibit SN4L to meet this goal. Consider the following sequence of demanded blocks: $..., A, A+1, A+2, B, B+1, C, D, ...$

On block A, SN4L prefetches $A+1$ and $A+2$. Moreover, Dis prefetcher is triggered on a prefetch for $A+2$ to prefetch B. However, SN4L and Dis prefetchers cannot make further progress, and the remaining blocks of this sequence remain non-prefetched. It means that block $B+1$ will be prefetched whenever processor demands block B. As a result, SN4L will prefetch $B+1$ with the effective depth of one that is not timely, as already discussed. To address this problem, we improve our SN4L and Dis prefetchers to prefetch the sequential region of detected discontinuities. As a result, when Dis prefetches B, SN4L attempts to prefetch $B+1$. Moreover, as Dis prefetcher is triggered on both fetch and prefetches, when SN4L prefetches B, Dis looks up DisTable to find discontinuities for blocks B and $B+1$. Then, DisTable finds the discontinuity of block $B+1$, and accordingly, prefetches C. This Dis prefetching triggers another SN4L prefetching. However, SN4L does not find any useful sequential prefetch for block C. Nevertheless, Dis prefetcher finds out that block D is a discontinuity prefetch candidate for block C.

This example clearly shows that it is possible to build a proactive prefetcher that can go multiple sequential and discontinuity regions ahead of the fetch stream, as far as needed. Such a possibility helps the prefetcher to issue timely prefetches. However, proactive SN4L and Dis prefetchers may trigger multiple SN4L and Dis prefetches, and each one may need to look up the local prefetch status bits, SeqTable, and DisTable. As a result, we need to have queues to accommodate the blocks that will trigger SN4L and Dis prefetchers (called *SeqQueue* and *DisQueue*, respectively). For example, in the given sequence, when SN4L finds $A+1$ and $A+2$ are useful sequential prefetches, it pushes these blocks to the end of *DisQueue* to trigger Dis prefetcher for these blocks. Consequently, SN4L and Dis prefetchers are triggered based on the block that is at the head of their corresponding queue. Moreover, such a chain of prefetches can generally have no limits. To avoid useless prefetches because of going far ahead of the fetch stream, this chain should be terminated at a point. We associate a depth to each block that is pushed to *SeqQueue* and *DisQueue*. In the beginning, proactive prefetching is triggered on a demanded block, and the rest of the prefetches are triggered based on the prior prefetch candidates. The associated depth for the first triggering block is zero. To set the depth of the remaining blocks, we add one to the depth of the triggering block that resulted in the prefetch. To clarify this, in the above example, in the first step, A is sent to *SeqQueue*, and its depth is zero. SN4L detects $A+1$ and $A+2$ are useful blocks. Now, these blocks are candidates to find Dis prefetches. As a result, $A+1$ and $A+2$ should be sent to *DisQueue*. As the depth of the block that has triggered their prefetch is zero, the depth of their associated entry in *DisQueue* will be one. Dis prefetcher finds B as a new prefetch candidate. In this step, B should be sent to *DisQueue* and *SeqQueue*. Similarly, the depth of block B in these queues will be two. This mechanism makes it possible to terminate a chain of prefetches when it reaches a specific depth. This proactive prefetching mechanism will be called SN4L+Dis prefetcher.

Our experiments show that *four* is a reasonable threshold to terminate the prefetch chain. Moreover, by going far ahead of the fetch stream, the timeliness is obtained at the cost of lower prefetch accuracy. As a result, we use SN1L, instead of SN4L, to prefetch the sequential regions of discontinuities.

Decreasing the unnecessary cache lookups: An aggressive prefetcher like SN4L+Dis issues lots of repetitive and unnecessary cache lookups. To avoid this, SN4L+Dis uses a simple structure named *Recently Looked Up (RLU)* that stores the address of the last eight blocks that most recently are looked up, either by the prefetcher or demanded by the processor. Every prefetch candidate that is determined by SN4L+Dis is sent to a queue named *RLUQueue*. Prefetch candidates are popped from this queue and looked up in RLU. If they miss in RLU, the prefetcher looks up the cache. If not found in the cache, the prefetch candidate will be sent to the memory hierarchy. Otherwise, the prefetch candidate is ignored.

The proactive mechanism makes progress whenever a new prefetch candidate is determined that is not in RLU. Consequently, an RLU miss is the actual event that pushes a new triggering block to *SeqQueue* and *DisQueue*. Therefore, it is necessary to have the depth of blocks in RLU. In other words, whenever SN4L+Dis finds a prefetch candidate, it not only sends the prefetch candidate to *RLUQueue* but also sends the depth of the block that has triggered the prefetch. Having the depth in *RLUQueue*, SN4L+Dis can push the new triggering blocks with the appropriate depths to *SeqQueue* and *DisQueue*.

[3] We decode the last two instructions because of the branch delay slot in the SPARC architecture.

C. BTB Prefetcher

To address the frequent BTB-miss problem, we borrow insights from Confluence [16]. However, as our goal is not to change the structure of BTB, we use a conventional program-counter based BTB. Instead, we prefetch to a BTB prefetch buffer and use a more aggressive prefetch mechanism to fill it. Unlike Confluence, we send cache blocks to the pre-decoder every time that they are missed in RLU. Note that as both Dis and BTB prefetchers need a pre-decoder, a single pre-decoder is used for both prefetchers. This brings the opportunity to do discontinuity and BTB prefetching simultaneously. To exploit this opportunity, Dis prefetcher looks up DisTable to find the discontinuity offset of the block that is at the head of DisQueue. Then the discontinuity offset is sent to the pre-decoder along with the instructions of that block. Predecoder decodes all the instruction of that block and extracts the branches and checks whether the instruction in the given offset is a branch or not, and extracts the target if it is a branch. Finally, this target is pushed to RLUQueue for the next steps.

Extracted branches will be stored in a prefetch buffer next to the BTB. We organize the BTB prefetch buffer entries similar to BTB entries of Confluence. This organization helps the BTB prefetcher to store all branches of a given cache block in just a single access to the BTB prefetch buffer without requiring any change to the BTB itself. The BTB prefetch buffer is a 2-way set-associative structure. A hit in the BTB prefetch buffer will send the appropriate entry to the BTB.

Compared to Confluence and Shotgun, our BTB prefetcher has some advantages. As compared to Confluence, the design is independent of the BTB type. Consequently, it can be used along with any BTB organization. As compared to Shotgun, our proposal is more straightforward. Shotgun uses a basic-block oriented BTB. It means that not only the branch instructions but also basic-block boundaries must be determined. Note that basic blocks may span multiple cache blocks. Consequently, the pre-decoder must keep a record of what it has already seen because cache blocks are fed one by one. Furthermore, Shotgun has two distinct proactive and reactive BTB prefilling mechanisms. As such, it must either have two separate pre-decoders or on a BTB miss that triggers reactive BTB prefilling, save the proactive BTB prefilling state to continue the process at the end of reactive BTB prefilling.

D. Handling Variable-Length ISA

Supporting machines with variable-length ISA (VL-ISA), some minor modifications are needed in our proposal. SN4L does not require any changes. Dis prefetcher needs to store the byte-offset instead of the instruction-offset in each DisTable entry. This way, the pre-decoder knows where the discontinuity branch instruction begins. This change increases the storage requirement of an entry from eight bits to ten bits (assuming a 64-byte cache block), resulting in a 20% increase in the storage cost of DisTable.

BTB prefetcher is the component that needs more consideration. Based on Figure 8, for each instruction block, a branch-footprint (BF) holding four byte-offsets is required to cover all branches. As devoting dedicated storage for this purpose imposes a high storage cost, we virtualize BFs in the LLC. SHIFT [21] already did LLC virtualization to store the instruction prefetcher's metadata. However, their implementation has two shortcomings. First, it loses a fraction of LLC all the time, even if we do not use it. Second, every LLC tag array needs to be extended to store the corresponding index to the virtualized sets, imposing 960 KB of extra storage to the LLC. On the contrary, our virtualization proposal, called dynamically-virtualized LLC (DV-LLC), is a dynamic mechanism that overcomes both problems.

DV-LLC switches between two modes: the least recently used (LRU) way is either a block-holder, as in conventional LLCs, or a BF-holder. When a cache set contains no instruction block, all ways are block-holders as in a conventional LLC. Otherwise, the LRU way is a BF-holder. When an instruction block is inserted into a set that does not have any instruction blocks, the LRU block is evicted (if utilized) for the LRU way to become a BF-holder. Moreover, when all instruction blocks within a cache set are evicted, the LRU way returns to its block-holder mode. For this purpose, every cache block has a single bit named *isInstruction*. Based on logical OR of all *isInstruction* bits in a set, the operation mode of the LRU way is determined. This implementation imposes less than 0.2% storage cost to a 32 MB LLC.

Moreover, as each BF is 3 bytes, a 64-byte block is sufficient to store the BF of up to 21 ways. If the associativity of LLC is less than 21, to ease finding the BF, we consider a design in which each way's BF is directly mapped to a specific location of the LRU data array. Consequently, no tagging is needed to determine the owner of the BF in the LRU way. However, if the associativity exceeds 21, the virtualized way can be looked up in a fully-associative manner. This means that the corresponding instruction block's tag should be included as well as the BF in the LRU way. Considering the tag overhead, the fully associative BF-holder can store the BF of up to ten instruction blocks. Fortunately, based on Figure 9, this is more than what we need to cover all BFs.

E. Example

Figure 10 provides an overview of the components of the proposed prefetcher. Suppose that block *A* is not in the cache. In consequence, access to this block will trigger a fetch request. Moreover, SeqTable is looked up to determine the prefetch status of its four subsequent blocks, *A+1* to *A+4*. From the figure, we see that these status bits are 0, 1, 0, and 1, respectively. As the prefetch status of *A+1* and *A+3* is 0, SN4L prefetcher only looks up *A+2* and *A+4* in the RLU. RLU filters *A+2* as it is already looked up in the cache. As *A+4* is not in the cache, a prefetch request for it will be sent. When block *A* arrives at the cache, the local prefetch status bits are placed near the block to ease accessing them in the future. Moreover, the DisTable is looked up, and a partial-tag match is detected for block *A*. Then, the block is sent to the pre-decoder to extract the branch instructions and prefill the BTB prefetch buffer. Besides, as the corresponding offset in the DisTable is 9, the pre-decoder checks if the ninth instruction is a branch. If the instruction is a branch, the pre-decoder calculates the target (Block *C* in our example). Then the target is looked up in the RLU. As the target (i.e., Block *C*) is not in the RLU and the cache, a prefetch request will be sent for it.

F. Comparison to Prior Work

Table II summarizes the comparison of our proposal and the state-of-the-art temporal prefetcher, Confluence [16] and the state-of-the-art BTB-directed prefetcher, Shotgun [20].

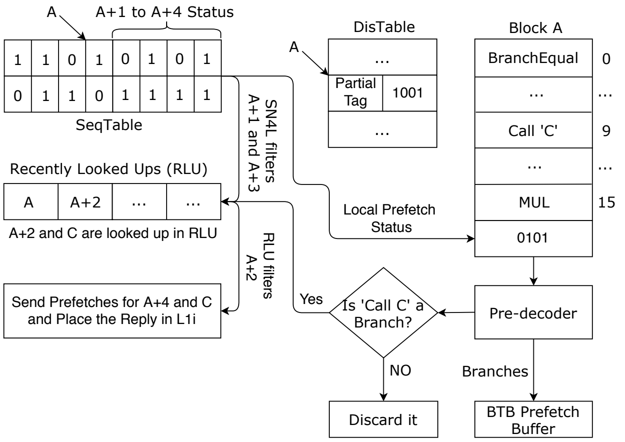

Fig. 10. Overview of the proposed prefetcher.

TABLE II
SN4L+DIS+BTB AND PRIOR WORK.

	SN4L+Dis+BTB	Shotgun [20]	Confluence [16]
Storage Overhead	7.6 KB	6 KB	1 MB overhead modifying LLC over 200 KB metadata virtualized in LLC
BTB modification	No	Yes	Yes
Instruction Prefetch Buffer	No	Yes	No
Scalability	6 KB	20 KB	-
Search Complexity	Low	High	High
Modularity	Yes	No	No
Handling Large Workloads	Yes	No	Yes

SN4L+Dis+BTB and Shotgun need small storage requirements, which are negligible as compared to that of Confluence.

While Shotgun needs a basic-block oriented BTB to work correctly and Confluence uses AirBTB, SN4L+Dis+BTB is independent of the BTB structure and can be used along with all BTB structures. Moreover, Shotgun uses a 64-entry instruction prefetch buffer to store the prefetches, but SN4L+Dis+BTB and Confluence prefetch directly into the L1i.

SN4L+Dis+BTB is more scalable than Shotgun. By allocating an extra 6 KB storage (the sum of storage needed for SN4L and Dis prefetchers), SN4L+Dis+BTB can store twice as much metadata to handle larger workloads. In comparison, Shotgun needs to double the U-BTB structure, which imposes about 20 KB storage overhead.

SN4L+Dis+BTB has the lowest search complexity because it looks up two direct-mapped structures that are kept near L1i. On the other hand, Confluence follows two steps to find prefetch candidates. First, it looks up the LLC to find the value of the pointer to the history buffer, which determines where the appropriate entry in the history buffer is and then looks up the LLC again using that pointer to find the spatial regions to extract the prefetch candidates. As the metadata of Confluence is kept in the LLC, the read-and-write process is complicated. In the case of Shotgun, it uses a 64-entry L1i prefetch buffer and a 32-entry BTB prefetch buffer to store prefetched candidates. These prefetch buffers have fully-associate structures, which impose significant search overhead as compared to SN4L+Dis+BTB. Moreover, Shotgun searches three different BTB structures, with various associativities, simultaneously, in each lookup.

Unlike other prefetchers, SN4L+Dis+BTB has a modular design: SN4L can be implemented independently from other components to achieve considerably better performance. Moreover, because the pre-decoding scheme is widely used in prior work to eliminate BTB misses, the Dis prefetcher requires low modification as it can reuse the pre-decoder.

Finally, SN4L+Dis+BTB and Confluence can handle immensely larger workloads, which is Shotgun's shortcoming. Generally, prefetchers' effectiveness depends on their ability to (1) maintain their metadata, and (2) lose the smallest opportunity in case of lack of metadata. Both SN4L+Dis+BTB and Confluence have devoted tables with a large number of entries to hold their metadata. On the other hand, Shotgun builds its prefetcher on top of a 1.5 K-entry U-BTB, which is small for large workloads. Moreover, a metadata miss in SN4L+Dis+BTB will result in a single instruction miss. Moreover, once an instruction miss occurs, SN4L and Dis prefetchers are instantly triggered to cover the probable instruction misses in that spatial location. The lack of metadata in the competing approaches is more significant than our proposal. Both proposals use footprints to prefetch the instructions in the spatial regions, and a footprint miss results in losing the ability to prefetch in the whole corresponding region. The consequence of Shotgun's U-BTB footprint misses is discussed in Section III.

VI. METHODOLOGY

Table III summarizes the key elements of our methodology, with the following sections detailing the specifics of the evaluated designs, workloads, prefetchers' configurations, and simulation infrastructure.

TABLE III
EVALUATION PARAMETERS.

Parameter	**Value**
Processing Nodes	14 nm, UltraSPARC III ISA, Sixteen 2 GHz OoO cores 3-wide dispatch/retirement, 128-entry ROB
Instruction Fetch Unit	32 KB, 8-way, 64B block, 4-cycle load-to-use 32-entry pre-dispatch queue TAGE [25] branch predictor, 2 K-entry Branch Target Buffer
L1d Cache	32 KB, 8-way, 64B block, 4-cycle load-to-use, 32 MSHRs
Shared LLC	32 MB, 16-way, 16 banks, 18 cycles access latency
Network-on-Chip	4×4 2D Mesh, Router: 5 ports, 3 VCs/port, 5 flits/VC 2-stage speculative pipeline, 1-cycle link traversal
Main Memory	60 ns access latency, 85 GB/s peak bandwidth

A. CMP Parameters

We model a server processor that resembles Intel Xeon Processor E7-4850 v4 [26]. The server processor has 16 cores and 32 MB of last-level cache (LLC). Four DDR4 memory channels provide up to 85 GB/s bandwidth. Each core is a three-way out-of-order design with 128-entry re-order buffer (ROB), 64-entry load-store queue (LSQ), and 32 KB 8-way set-associative L1i and L1d caches.

The cores are organized in a 4-by-4 grid of 16 tiles. Each tile contains a core, a slice of the LLC, and a directory slice. A mesh interconnect is used to connect tiles. At each hop, a packet goes to a two-stage router pipeline followed by a single-cycle link traversal for a total of three cycles per hop at zero loads.

For L1i prefetching, we assume that each L1i cache has two ports to perform up to two cache lookups per cycle. Moreover, a 32-entry prefetch queue is used to hold potential prefetch candidates before performing L1i cache lookups.

The simulated cores have three fetch stages in the core frontend and 12 pipeline stage in the backend. The wrong-path execution as a result of BTB misses and branch mispredictions are modeled. The pipeline squashes and correct-path redirections are done in the third stage of the core backend, resulting in at least a six-cycle penalty, depending on when the instruction arrives in the appropriate pipeline stage.

B. Workloads

We simulate workloads that are listed in Table IV. We include a variety of server workloads from competing vendors, including online transaction processing, cloud web serving system, streaming server, and web server benchmarks.

TABLE IV
SERVER WORKLOADS.

OLTP - Online Transaction Processing (TPC-C)	
DB A	Oracle 10g Enterprise Database Server 100 Warehouses (10 GB), 1.4 GB SGA
DB B	IBM DB2 v8 ESE Database Server 100 Warehouses (10 GB), 2 GB Buffer Pool
Web Server (SPECweb99)	
Apache	Apache HTTP Server v2.0 16 K Connections, fastCGI, Worker Threading
Zeus	Zeus Web Server v4.3 16 K Connections, fastCGI
CloudSuite	
Media Streaming	Darwin Streaming Server 6.0.3 7500 Clients, 60 GB Dataset, High Bitrates
Web Frontend	Nginx 1.0.10 Built-in PHP 5.3.5 & APC 3.1.8
Web Search	Nutch 1.2/Lucene 3.0.1, 230 Clients 1.4 GB Index, 15 GB Data Segment

C. Simulation Infrastructure

We estimate server workloads on a 16-core server processor using Flexus full-system simulator [27]. Flexus extends the Virtutech Simics functional simulator with timing models of cores, caches, on-chip protocol controllers, and interconnect. Flexus models the SPARC v9 ISA and is able to run unmodified operating systems and applications.

We use the SimFlex multiprocessor sampling methodology [28] that extends the SMARTS sampling framework [29] for multiprocessors. We have more than 100 checkpoints per workloads, where a checkpoint includes architectural and long-term microarchitectural states like cache and branch predictor contents. For each workload, we launch simulations from checkpoints and run 200 K cycles to warm short-term microarchitectural states like re-order buffer and then collect measurements for the subsequent 200 K cycles. Performance measurements are computed with a 95% confidence level and a confidence interval of less than 4%.

D. Prefetchers' Configurations

We evaluate the following prefetchers.

1) Confluence: Confluence [16] is the state-of-the-art temporal prefetcher that offers a unified solution for instruction prefetching and BTB prefilling. Confluence uses SHIFT [21] as the instruction prefetch engine and virtualizes the metadata in the last-level cache (LLC). It also pre-decodes the prefetched blocks to fill the missing BTB entries. We model Confluence as SHIFT, and a 16 K-entry BTB is used along with it. It has been shown that this design offers an upper bound for what can be achieved by Confluence [16].

2) Shotgun: Shotgun [20] is the state-of-the-art BTB-directed prefetcher. It benefits from Boomerang [19] for L1i and BTB prefetching. Moreover, Shotgun uses dedicated BTBs for the unconditional branch, conditional branch, and return instructions. Shotgun uses a 32-entry FTQ, a fully-associative 32-entry BTB prefetch buffer, and a fully-associative 64-entry L1i prefetch buffer. We evaluate a 1.5 K-entry U-BTB, a 128-entry C-BTB, and a 512-entry RIB, as has been suggested by the original proposal. The total per-core storage overhead of Shotgun is 6 KB, which is due to additional segments in the BTB to store basic-block length and footprints, and the L1i and BTB prefetch buffers.

3) SN4L+Dis+BTB: SN4L+Dis+BTB uses a simple and accurate sequential prefetcher named SN4L and augments it with a lightweight discontinuity prefetcher named Dis and pre-decodes the touched blocks to prefill the missing BTB entries. SN4L uses a 16 K-entry direct-mapped and tagless table, which needs 2 KB storage, Dis uses a 4 K-entry direct-mapped and 4-bit partially-tagged table, which requires 4 KB storage and a 32-entry, Confluence-like BTB prefetch buffer, which is organized as a 2-way set-associative table, which imposes 1 KB storage overhead. Moreover, we need a 4-bit local prefetch status and 1-bit *isPrefetch* flag for each block in the cache. Besides, SeqQueue, DisQueue, and RLUQueue have 16 entries and are used along with an 8-entry RLU, which together require 0.3 KB. The overall storage overhead of SN4L+Dis+BTB is 7.6 KB.

VII. EVALUATION RESULTS

A. Storage Requirements

The proposed prefetching technique needs two history tables: (1) SeqTable and (2) DisTable. We organize both tables as direct-mapped structures. Empirically, we observe that such simple structures perform well within the context of our proposal. Figure 11 shows how the miss coverage of SN4L changes when the number of entries that are dedicated to SeqTable varies. We also measure the miss coverage for the unlimited-size table as a point of reference. In the unlimited table, a block has its dedicated entry. As the table becomes larger, the miss coverage increases. With 16 K entries, however, the coverage reaches 96% of the unlimited table's, effectively exploiting the available opportunity. Therefore, we decide to devote 16 K entries to the SeqTable, which requires 2 KB of storage.

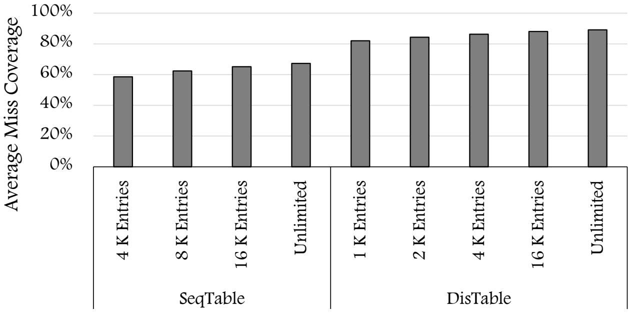

Fig. 11. Miss coverage for various table sizes.

Determining the size of SeqTable, Figure 11 shows the coverage of SN4L+Dis for various numbers of entries in

DisTable. SeqTable has 16 K entries. The unlimited table is also reported in which each block has its dedicated entry. Like SeqTable, as the size of DisTable increases, the prefetcher covers more cache misses. As the coverage reaches the 97% of its maximum value when DisTable has 4 K entries, we choose a 4 K-entry DisTable. SeqTable and DisTable together need 6 KB storage.

B. Tables' Associativity

Figure 11 reveals that the direct-mapped SeqTable and DisTable offer 96% of the coverage of the unlimited tables in which each block has a dedicated place and never conflicts with other blocks. As such, we do not use set-associative structures for these tables.

C. Tagless and Partially-Tagged Policy

As discussed in Section V, SN4L uses a tagless table and Dis benefits from a 4-bit partial tag to avoid conflicts, and hence, useless prefetches. In this section, we show why we do not need the full tag for these two tables. With SN4L, the size of SeqTable is quite large and has 16 K entries, and each entry is just a single bit. Generally, a conflict in the table results in a correct prediction with 0.5 probability. Our experiments indicate that the conflict ratio of the table is 28%, and the table makes correct predictions in 92% of the times. Consequently, a tagless table is sufficient.

DisTable, on the other hand, is a 4-bit partially-tagged table. While Figure 11 shows that there is a negligible difference between a 4 K-entry table and an unlimited one in terms of miss coverage, Figure 12 shows the overprediction when different tagging policies are used. The figure reveals that while a tagless table suffers from a considerable overprediction, a 4-bit partially-tagged table moderates the overprediction as compared to a fully-tagged table. In consequence, DisTable is a 4-bit partially-tagged table to reduce the overprediction significantly.

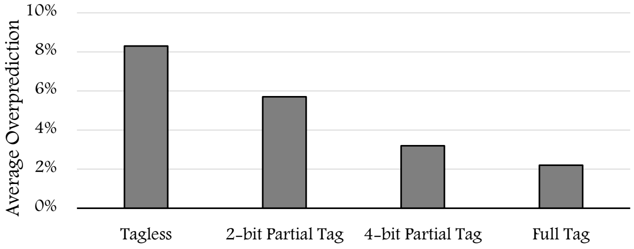

Fig. 12. Overprediction of different tagging policies.

D. Prefetch Timeliness

A useful prefetcher should provide sufficient timeliness for its prefetching requests. Figure 13 shows the average timeliness in terms of the covered memory access latency (CMAL) for the N4L, SN4L, Dis, and SN4L+Dis+BTB prefetchers. SN4L offers 93% CMAL, 5% better than N4L. As the prefetch depth is the same for both prefetchers, SN4L's better timeliness is due to its lower traffic, which reduces the average LLC access latency. Dis' CMAL is 89%, which is lower than that of SN4L because of its larger path to issue prefetches, which includes DisTable lookup and pre-decoding. Finally, SN4L+Dis+BTB covers 91% of the memory access latency. One might expect that SN4L+Dis+BTB's CMAL to be close to SN4L's because SN4L contributes to a larger fraction of prefetches as compared to Dis. However, the reduction in SN4L+Dis+BTB's CMAL as compared to SN4L is because of having a BTB prefetcher. By keeping the execution on the correct-path, demand instructions are issued faster, which results in more uncovered clock cycles in the case of untimely prefetches.

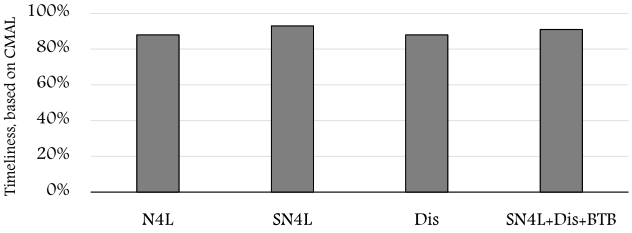

Fig. 13. Timeliness of different prefetchers.

E. RLU Size and Cache Lookups

The *recently looked up* policy is used to filter useless cache lookups of our proposal. Figure 14 shows the number of cache lookups of several methods, normalized to the number of cache lookups in a processor without a prefetcher. The results clearly show that an RLU of eight entries performs well in the context of our proposal. Confluence offers the lowest number of cache lookups among the evaluated approaches. Our proposal and Shotgun require the same number of cache lookups to perform effectively.

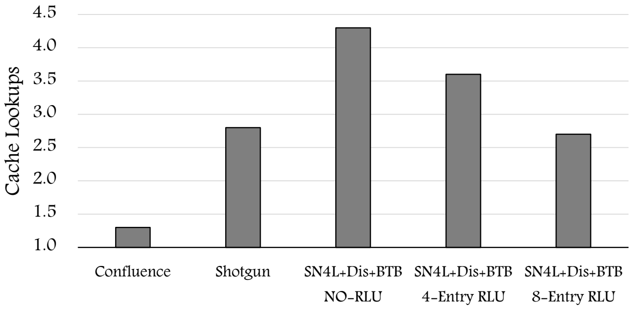

Fig. 14. Number of cache lookups, normalized to baseline with no prefetcher.

F. Frontend Stall Cycle Reduction

To evaluate the effectiveness of the proposed prefetcher, Figure 15 shows the Frontend Stall Cycle Reduction (FSCR) for each method. FSCR of each method is the fraction of L1i-/BTB-induced stall cycles that are eliminated by the method. FSCR, unlike commonly-used miss coverage, captures the impact of on-the-fly prefetch requests[4], and more accurately shows the ability of each method to mitigate frontend-induced stalls.

There are two reasons for the frontend stall cycles. First, a core cannot make progress because the demanded instruction was missing, and it had to wait until the requested block

[4] Requests that have been issued by the prefetcher, but the corresponding cache blocks have not arrived at the fetch unit.

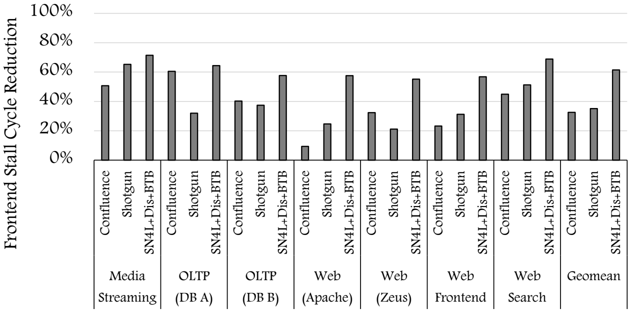

Fig. 15. Frontend Stall Cycle Reduction (FSCR) of the evaluated methods. The more FSCR a method exhibits, the more frontend-induced stalls are captured by the method.

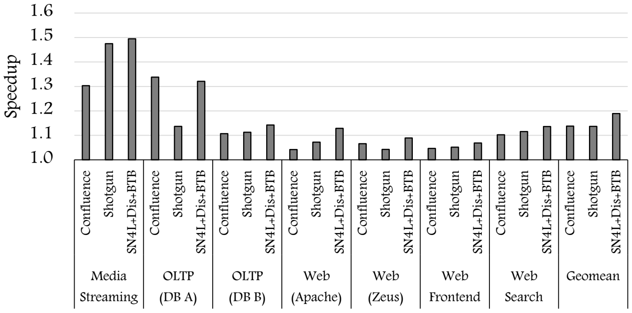

Fig. 16. Performance comparison of evaluated methods, normalized to a baseline system with no instruction/BTB prefetcher.

becomes available. Second, in BTB-directed prefetchers, a stall cycle can also be the consequence of a BTB-miss. As BTB misses stall the address generator that is feeding the cache with new addresses, when a core consumes all of the available instructions, it should wait until the BTB miss is resolved and the address generator begins its operation. Approaches that are not based on BTB-directed prefetching do not suffer from this type of frontend stall cycles.

SN4L+Dis+BTB offers the highest FSCR among the competing approaches by covering 61% of the frontend stall cycles. Shotgun and Confluence stand in the second and third places by covering 35 and 32% of the frontend stall cycles, on average.

G. Speedup

Figure 16 shows the performance improvement of our proposed method along with other techniques, over a baseline without any instruction/BTB prefetcher. The performance improvement of SN4L+Dis+BTB ranges from 7% in Web Frontend to 50% in Media Streaming.

Our proposal, SN4L+Dis+BTB, significantly improves the performance and outperforms the competing techniques. On average, SN4L+Dis+BTB enhances the performance by 19%, which is 5% higher than Shotgun, the recent state-of-the-art proposal. Moreover, on the OLTP (DB A), which has the highest U-BTB footprint miss ratio, the performance improvement of our proposal over Shotgun is 16%.

It can be seen that Confluence outperforms SN4L+Dis+BTB in OLTP (DB A). The main reason is that our Confluence, instead of BTB prefetching, uses a 16 K-entry BTB, which is shown to offer near-ideal BTB performance [16]. As a result, our Confluence shows an upper bound of its performance. We used such a policy because Confluence with 16 K-entry BTB still lags behind SN4L+Dis+BTB in overall performance. Note that while SN4L+Dis+BTB has better FSCR than Confluence on OLTP (DB A), it has a lower speedup. This means that SN4L+Dis+BTB is active in the wrong-path because it cannot direct the execution stream to the correct-path because of uncovered BTB misses.

H. Performance Breakdown

Figure 17 shows the performance breakdown of SN4L+Dis+BTB. As a point of reference, we include a *Perfect L1i* in which all requests are served with the delay of a cache hit, and a *Perfect L1i + BTB*∞ in which a 32 K-entry BTB is used along with the *Perfect L1i*. As compared to N4L, SN4L offers 5% speedup, which signifies the importance of selective prefetching. SN4L and SN4L+Dis offer 13 and 15% performance improvement over a system with no instruction and BTB prefetcher. SN4L+Dis+BTB reaches 19% speedup, which is only slightly lower than that of *Perfect L1i*. With *Perfect L1i + BTB*∞, the performance improvement becomes 29%, which is 10% higher than that of our practical proposal.

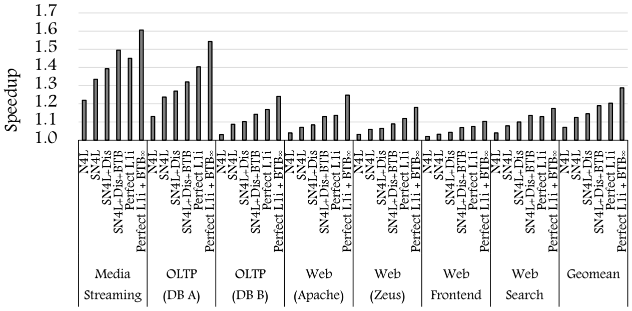

Fig. 17. Performance breakdown of SN4L+Dis+BTB and comparison to perfect frontend.

I. Large Workloads

Recent studies show that commercial server workloads have larger instruction footprints and higher BTB miss ratio as compared to server benchmarks [1], [5]. To study the effect of commercial server workloads with larger instruction footprints on instruction prefetchers, we make the BTB smaller and assess Shotgun and SN4L+Dis+BTB prefetchers. Figure 18 shows the average speedup of our proposal over Shotgun across the evaluated workloads. The figure clearly shows that as the BTB size decreases, and hence, the BTB miss ratio increases, the gap between our proposal and Shotgun increases. While in this experiment, we increase the BTB miss ratio, and as such, BTB prefetcher becomes more valuable, the instruction footprints of workloads remain the same. Consequently, we do not put extra pressure on the L1i cache. With commercial server workloads, there is more pressure on both the L1i and BTB. As such, we expect a wider gap between SN4L+Dis+BTB and Shotgun.

J. Variable-Length ISA

While we showed that the proposed mechanism to handle the variable-length ISA provides the ability to capture almost

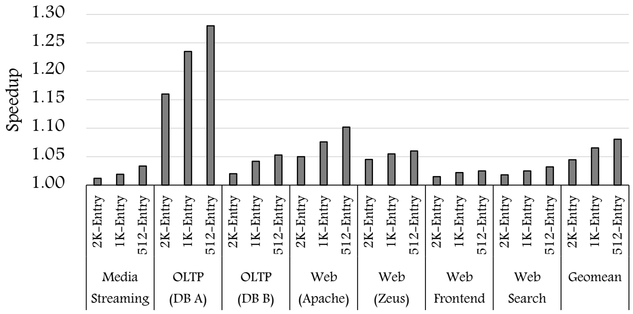

Fig. 18. Speedup of our proposal over Shotgun with varying BTB sizes.

all of branches, virtualizing the branch-footprints inside the LRU way may decrease the effective LLC capacity and offset the benefits. Our experiments show that the DV-LLC remains as effective as a conventional LLC. The instruction hit ratio of the LLC remains the same, while the data hit ratio, in the worst case, drops 0.1%. While our simulated machine has fixed-length ISA, we also evaluated the BTB prefetcher using the DV-LLC. As expected, the same speedup is achieved (the graph is not shown due to space limitation).

VIII. Related Work

Frontend stalls are a fundamental performance bottleneck in servers [2], [30]–[36]. Deep software stack of server workloads and heavily relying on operating-system functionalities synergistically lead to vast instruction footprints that overwhelm capacity-limited instruction caches and branch target buffers. Next-line prefetchers [8], which are used in most commercial processors, cover only a small fraction of instruction misses, resulting in a significant performance loss. As a consequence, a myriad of proposals attempted to mitigate the negative effects of frontend stalls by using sophisticated prefetchers [9]–[12], [14]–[21], [37]–[40].

In the software side, strategies were proposed to reduce the number of instruction misses either by inserting compile-time prefetch requests [38], [39], or optimizing the application code for a higher locality [41]–[43], or predicting prefetch addresses using the recurring call-graph history [37]. These techniques make the instruction sequences more predictable, which enables our proposal to offer either higher miss coverage or the same miss coverage with smaller storage.

Branch predictor directed prefetchers use existing branch predictors to explore the program's control-flow graph ahead of the fetch unit to predict discontinuities along the expected path [9]–[11]. Run-ahead execution [44], [45] and speculative threading [46], [47] likewise use branch predictors to predict future control flow of a program for prefetching. Even though these schemes do not require extra storage, they heavily depend on the accuracy of branch predictors; moreover, they do nothing for BTB-induced stalls.

Streaming techniques like TIFS [14] and PIF [15] record, index, and prefetch an arbitrary-length instruction miss/access history to overcome the limitations of branch predictor directed prefetchers. While these techniques are effective at reducing fetch-related stalls, they require excessive storage for logging cache misses/accesses.

SHIFT [21] and Confluence [16] exploit the commonality of instruction streams among cores when they run the same workload and amortize the cost of the entire history storage across multiple cores by adopting a shared history, virtualized in the LLC. Unfortunately, the efficiency of these techniques over stream-based methods is limited by the number of different workloads running on a processor. Whenever multiple workloads share a processor, each workload requires its metadata and places additional pressure on the LLC and the interconnect, which may offset the benefits.

IX. Conclusion

Processors frequently encounter instruction, and BTB misses on server workloads. As these misses often stall processors, server workloads lose performance due to inefficient instruction supply. Unfortunately, next-line prefetchers, which are available in processors, fall short of efficiency at eliminating many misses. While prior temporal instruction prefetchers can hide the latency of instruction misses, they require excessive storage for the metadata. The recent BTB-directed instruction prefetchers impose low overhead but require significant changes in the frontend of processors and are less suitable for workloads with enormous instruction footprints.

In this work, we offered a divide-and-conquer approach to address the frontend bottleneck. We showed that while a next-four-line (N4L) prefetcher can eliminate most of the sequential instruction misses, it produces a large number of useless prefetches. We showed that useless prefetches of an N4L prefetcher is predictable and can easily be eliminated. Moreover, we showed that a discontinuity prefetcher could eliminate the remaining non-sequential misses. We benefited from the fact that discontinuities are a consequence of branch instructions to reduce the area overhead of the discontinuity prefetcher. Finally, we used a BTB prefetcher to eliminate BTB misses. The proposed prefetcher requires 7.6 KB of storage and boosts performance by up to 16% over the most-recent competitor.

Acknowledgment

The authors would like to thank the anonymous reviewers for their feedback and suggestions. We thank Mohammad Bakhshalipour for his assistance with the earlier draft of this paper. We thank IPM HPC center and Turin Cloud Services (turin.ipm.ir) for providing the compute-capabilities to carry out the experiments. This work is supported in part by a grant from the Iran National Science Foundation (INSF).

References

[1] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a Warehouse-scale Computer," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2015, pp. 158–169.

[2] N. Hardavellas, I. Pandis, R. Johnson, N. G. Mancheril, A. Ailamaki, and B. Falsafi, "Database Servers on Chip Multiprocessors: Limitations and Opportunities," in *Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research (CIDR)*, Jan. 2007, pp. 79–87.

[3] K. Lim, P. Ranganathan, J. Chang, C. Patel, T. Mudge, and S. Reinhardt, "Understanding and Designing New Server Architectures for Emerging Warehouse-Computing Environments," in *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2008, pp. 315–326.

[4] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the Clouds: A Study of Emerging Scale-Out Workloads on Modern Hardware," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2012, pp. 37–48.

[5] G. Ayers, J. H. Ahn, C. Kozyrakis, and P. Ranganathan, "Memory Hierarchy for Web Search," in *Proceedings of the 24th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2018, pp. 643–656.

[6] M. Bakhshalipour, S. Tabaeiaghdaei, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Evaluation of Hardware Data Prefetchers on Server Processors," *ACM Computing Surveys (CSUR)*, vol. 52, no. 3, pp. 1–29, Jun. 2019.

[7] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer, and B. Falsafi, "Scale-Out Processors," in *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2012, pp. 500–511.

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

[9] A. V. Veidenbaum, "Instruction Cache Prefetching Using Multilevel Branch Prediction," in *Proceedings of the International Symposium of High-Performance Computing (ISHPC)*, Nov. 1997, pp. 51–70.

[10] 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.

[11] V. Srinivasan, E. S. Davidson, G. S. Tyson, M. J. Charney, and T. R. Puzak, "Branch History Guided Instruction Prefetching," in *Proceedings of the 7th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Jan. 2001, pp. 291–300.

[12] Y. Zhang, S. Haga, and R. Barua, "Execution History Guided Instruction Prefetching," in *Proceedings of the 16th Annual ACM International Conference on Supercomputing (ICS)*, Jun. 2002, pp. 199–208.

[13] A. Ramirez, O. J. Santana, J. L. Larriba-Pey, and M. Valero, "Fetching Instruction Streams," in *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, Nov. 2002, pp. 371–382.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] A. Kolli, A. Saidi, and T. F. Wenisch, "RDIP: Return-address-stack Directed Instruction Prefetching," in *Proceedings of the 46th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, Dec. 2013, pp. 260–271.

[19] 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.

[20] 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.

[21] 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.

[22] C. Xia and J. Torrellas, "Instruction Prefetching of Systems Codes with Layout Optimized for Reduced Cache Misses," in *Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA)*, May 1996, pp. 271–282.

[23] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Bingo Spatial Data Prefetcher," in *Proceedings of the 25th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2019, pp. 399–411.

[24] 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, Oct. 2018.

[25] A. Seznec and P. Michaud, "A Case for (Partially)-Tagged Geometric History Length Predictors," *Journal of Instruction Level Parallelism (JILP)*, 2006.

[26] Intel, "Intel Xeon Processor E7-4850 v4," https://ark.intel.com/products/93806/.

[27] Flexus, http://parsa.epfl.ch/simflex/flexus.html.

[28] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe, "SimFlex: Statistical Sampling of Computer System Simulation," *IEEE Micro*, vol. 26, no. 4, pp. 18–31, July-August 2006.

[29] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling," in *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2003, pp. 84–97.

[30] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood, "DBMSs on a Modern Processor: Where does Time Go?" in *Proceedings of the 25th International Conference on Very Large Data Bases (VLDP)*, Sep. 1999, pp. 266–277.

[31] S. M. Ajorpaz, E. Garza, S. Jindal, and D. A. Jiménez, "Exploring Predictive Replacement Policies for Instruction Cache and Branch Target Buffer," in *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2018, pp. 519–532.

[32] Q. Cao, P. Trancoso, J.-L. Larriba-Pey, J. Torrellas, R. Knighten, and Y. Won, "Detailed Characterization of a Quad Pentium Pro Server Running TPC-D," in *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, Oct. 1999, pp. 108–115.

[33] K. Keeton, D. A. Patterson, Y. Q. He, R. C. Raphael, and W. E. Baker, "Performance Characterization of a Quad Pentium Pro SMP Using OLTP Workloads," in *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA)*, Jun. 1998, pp. 15–26.

[34] J. L. Lo, L. A. Barroso, S. J. Eggers, K. Gharachorloo, H. M. Levy, and S. S. Parekh, "An Analysis of Database Workload Performance on Simultaneous Multithreaded Processors," in *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA)*, Jun. 1998, pp. 39–50.

[35] P. Ranganathan, K. Gharachorloo, S. V. Adve, and L. A. Barroso, "Performance of Database Workloads on Shared-Memory Systems with Out-of-Order Processors," in *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 1998, pp. 307–318.

[36] R. Stets, K. Gharachorloo, and L. Barroso, "A Detailed Comparison of Two Transaction Processing Workloads," in *Proceedings of the IEEE International Workshop on Workload Characterization (WWC)*, Nov. 2002, pp. 37–48.

[37] M. Annavaram, J. M. Patel, and E. S. Davidson, "Call Graph Prefetching for Database Applications," *ACM Transactions on Computer Systems (TOCS)*, vol. 21, no. 4, pp. 412–444, Nov. 2003.

[38] P. Kallurkar and S. R. Sarangi, "pTask: A Smart Prefetching Scheme for OS Intensive Applications," in *Proceedings of the 49th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, Oct. 2016, pp. 3:1–3:12.

[39] C.-K. Luk and T. C. Mowry, "Cooperative Prefetching: Compiler and Hardware Support for Effective Instruction Prefetching in Modern Processors," in *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, Nov. 1998, pp. 182–194.

[40] J. Yan and W. Zhang, "Analyzing the Worst-case Execution Time for Instruction Caches with Prefetching," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 8, no. 1, pp. 7:1–7:19, Jan. 2009.

[41] S. Harizopoulos and A. Ailamaki, "Steps Towards Cache-resident Transaction Processing," in *Proceedings of the 13th International Conference on Very Large Data Bases (VLDP) - Volume 30*, Sep. 2004, pp. 660–671.

[42] A. Ramirez, L. A. Barroso, K. Gharachorloo, R. Cohn, J. Larriba-Pey, P. G. Lowney, and M. Valero, "Code Layout Optimizations for Transaction Processing Workloads," in *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA)*, Jul. 2001, pp. 155–164.

[43] J. Zhou and K. A. Ross, "Buffering Database Operations for Enhanced Instruction Cache Performance," in *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, Jun. 2004, pp. 191–202.

[44] O. Mutlu, H. Kim, and Y. N. Patt, "Techniques for Efficient Processing in Runahead Execution Engines," in *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2005, pp. 370–381.

[45] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors," in *Proceedings of the 9th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2003, pp. 129–140.

[46] C.-K. Luk, "Tolerating Memory Latency Through Software-controlled Pre-execution in Simultaneous Multithreading Processors," in *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA)*, Jul. 2001, pp. 40–51.

[47] C. Zilles and G. Sohi, "Execution-based Prediction Using Speculative Slices," in *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA)*, Jul. 2001, pp. 2–13.