Synthesizing Code Prefetchers from Control Speculation Hardware

Mainak Chaudhuri
Indian Institute of Technology Kanpur

May 16, 2020

ABSTRACT

Timely instruction delivery is an important prerequisite for smooth progress of the instruction processing pipelines found in the microprocessors. Instruction cache misses can severely hamper this progress. As a result, code prefetchers play an important role in mitigating the bottlenecks encountered in the front-end of the instruction processing pipelines. A typical code prefetcher attempts to run significantly ahead of the demand instruction stream while fetching the code blocks that the upcoming control flow is likely to touch. Since the operations of a code prefetcher closely resemble those of a branch predictor, in this paper we propose to synthesize code prefetchers directly from the well-researched control speculation hardware. We first pose the code prefetching problem in such a way that it becomes equivalent to the control speculation problem and then design a code prefetcher that sports a partially tagged gshare predictor fused into a partially tagged prefetch target buffer, which closely resembles a traditional branch target buffer. Compared to a baseline that has no code prefetching at L1 cache, our proposal, evaluated on a single-core processor having a 32 KB 8-way L1 instruction cache, saves 83.7% instruction cache demand misses and achieves a speedup of 27.6% while offering an L1 instruction cache demand hit rate of 97.7% averaged over fifty code-intensive client, server, and SPEC CPU application traces. This performance comes close to the ideal oracle prefetcher that achieves a speedup of 30.6% while offering 100% L1 instruction cache hits for the demand stream.

1. INTRODUCTION

The instruction processing pipelines found in the microprocessors need to be fed with instructions at a certain rate for optimal performance. The L1 instruction cache is designed to deliver instructions at least at this rate. However, L1 instruction cache misses lower the average rate of instruction delivery leading to insertion of bubbles into the instruction processing pipelines. Code prefetching is an important technique that attempts to hide this inefficiency by running significantly ahead of the demand stream and prefetching code blocks that are likely to be touched in the near future. As a result, a typical code prefetcher is required to predict the upcoming instruction block addresses. Since this is very similar to what a branch predictor does at a finer grain of basic blocks (predicts the entry point of the next dynamic basic block on encountering a control transfer instruction), in this paper we propose to design an L1 cache code prefetcher that synthesizes its operations and storage structures directly from the vastly researched field of control speculation techniques.

We pose the code prefetching problem as a branch prediction problem at the grain of code cache blocks (as opposed to basic blocks). This opens up a large number of possibilities for synthesizing code prefetchers. In this study, we confine our exploration to a simple predictor that resembles a partially tagged branch target buffer (which we refer to as the prefetch target buffer) with an embedded partially tagged gshare predictor. To increase the effective reach of the prefetch target buffer, we couple the design with a simple dynamic prefetch target compression technique assisted by a small temporally populated dictionary. Our design also incorporates a simple prefetch filter to avoid prefetches to blocks that have recently been demanded or prefetched. We evaluate our proposal on a single-core system having a three-level cache hierarchy including a 32 KB 8-way L1 instruction cache on fifty code-intensive traces collected from client, server, and SPEC CPU applications. Our proposal, on average, saves 83.7% instruction cache demand misses and achieves a speedup of 27.6% relative to a baseline that does not have an L1 cache code prefetcher. Our proposal enjoys an L1 instruction cache demand hit rate of 97.7% on average. In comparison, the ideal oracle prefetcher, which offers 100% L1 instruction cache hit rate while accounting for all resource consumption (i.e., consumption of bandwidth and cache capacity) of the prefetcher throughout the entire memory hierarchy, achieves an average speedup of 30.6%. We also compare our proposal with a few existing ones, namely an optimized next-two-line prefetcher, the return-directed instruction prefetching (RDIP) technique, the proactive instruction prefetching (PIF) technique implemented with the instruction cache access order, and an optimized PIF technique; these proposals speed up execution by 12.5%, 18.8%, 21.4%, and 24.5% on average respectively. These and other related research contributions are discussed in the next section.

2. RELATED WORK

The next-line and next-*n*-line code prefetchers are the earliest code prefetchers proposed in the literature [22, 23]. These prefetchers rely on the sequential pattern of code execution. However, control transfer instructions introduce discontinuities in this sequential pattern. Several studies have approached this problem of discontinuities by correlating a sequence of prefetch targets with the current code block address [24], with the branch history patterns [26], or with the past code block addresses to improve the prefetcher's lookahead [25]. These techniques predict a sequence of prefetch addresses by remembering such correlations. As we will see in the next section, our proposal also attempts to predict the discontinuities in the sequential code stream, but appeals to the rich design space of control speculation hardware to do so.

Branch predictors that run ahead of the main execution stream and offer prefetch candidates with significant lookahead have been explored [3, 19]. Such branch-prediction-directed or fetch-directed prefetching techniques require the control speculation unit to be decoupled from the instruction fetch unit for achieving significant lookahead that is independent of the instruction cache performance. Lately, such decoupled prefetch units have been improved significantly by prefilling the branch target buffer (Boomerang) [15] or by short-circuiting branch prediction stages in the cases of pipeline flushes (elastic fetch) [18]. A recent proposal has further improved branch target buffer-directed prefetching with the help of a specialized branch target buffer design that maintains the global control flow across functions and a region-encoding capturing the local control flow within the functions (Shotgun) [16].

Temporal instruction fetch streaming (TIFS) [6] and proactive instruction fetching (PIF) [5] exploit the observation that the control flow paths repeat themselves in most server applications. However, since the repeat interval is very large, these proposals require large history buffers for recording the sequence of executed code regions of a fixed size (e.g., eight cache blocks); the specific cache blocks touched within a region are recorded in a bitvector. While TIFS records only the discontinuities in the instruction cache access stream, PIF records all touched code blocks in retire order. An index table is looked up using the current code region address and the index table entry points to the latest matching entry in the history buffer triggering code prefetches starting from that history buffer entry. Subsequently, shared history instruction fetch (SHIFT) has significantly reduced the metadata overhead of PIF by sharing the history buffer across cores and embedding the metadata in the large on-chip last-level cache [10]. Confluence unifies the metadata for prefetching into the L1 instruction cache and the branch target buffer by proposing a new branch target buffer organization, each entry of which maintains a bundle of branches as a bitvector corresponding to all branches in an instruction cache block [11]. Confluence further minimizes the metadata overhead by backing up the branch target buffer in the last-level cache following the idea of virtualized branch target buffer [1].

Return-directed instruction prefetching (RDIP) correlates the history of code regions with a hash of the top few entries in the return address stack [14]. This technique also incorporates lookahead into the prefetcher by correlating the code regions of the current function with a return address stack hash seen in the recent past. A similar approach involving function and event signatures has been used for prefetching in web applications [2].

Our proposal, to some extent, resembles those along the line of fetch-directed or branch prediction-directed prefetching. However, our proposal makes predictions at the grain of cache blocks. We employ partially tagged gshare and branch target buffer designs to offer these predictions. Partially tagged global history-based direction as well as target predictors have been studied in great detail. For example, the TAGE and ITTAGE predictors use multiple global predictor components to do this [20, 21]. The other studies on target prediction of indirect branches include devirtualization of indirect branch's instruction pointer [12] and correlating the output value of some hint instruction with the target of an indirect branch [4]. Recent studies have exploited an ensemble of neural predictors for target prediction of indirect branches [7]. Replacement and insertion policy optimizations for the instruction cache and the branch target buffer have also been studied [17]. All these design optimizations can be seamlessly applied to the branch target buffer proposed by us. In this study, we restrict ourselves to a simple design.

3. DESIGN OF PROPOSED PREFETCHER

We first reduce the code prefetching problem to a control flow speculation problem at block grain and then discuss the design of our proposed prefetcher.

3.1 Control Flow at a Coarse Grain

Let us consider an example instruction fetch stream that accesses the following sequence of cache block addresses: A, $A+1$, $A+2$, B, C, $C+1$, D, $D+1$, A,..., where B, C, D, A are the discontinuities from the sequential stream i.e., $B \neq A+3; C \neq B+1; D \neq C+2; A \neq D+2$. We imagine each block address to be a branch address. The branch is taken if the next block address in the sequence represents a discontinuity and the target of the branch is the next block address; otherwise the branch is not taken. In the above example, $A, A+1, C, D$ are not taken branches, while the rest are taken branches. The targets of the taken branches $A+2, B, C+1, D+1$ are respectively B, C, D, A i.e., precisely the set of discontinuities. We further define a block-grain global control flow history based on this taken/not taken behavior. In the above example, this block-grain global history would be 00110101 where 0 represents a not taken branch and 1 represents a taken branch. Therefore, the code prefetching problem is equivalent to predicting taken/not-taken for each code block and if the prediction is taken, a target prediction is also needed. With this reduction of the code prefetching problem to a block-grain control flow speculation problem, it is easy to see that one can now appeal to the large body of studies on control speculation to synthesize varieties of code prefetchers. We discuss one simple design in the following. We note that we will need a direction predictor (indicating taken/not taken) and a target predictor for taken branches.

The aforementioned technique offers the impression that it will work only when a code block has a single control transfer instruction and this instruction exhibits consistent taken/not-taken behavior. However, a code block can have multiple control transfer instructions and they can exhibit differing taken/not-taken behavior over time. To address this, we will use the block-grain global history along with the block address to train our predictor. We observe that given a particular block-grain global history pattern, a code block's taken/not-taken behavior is quite consistent over time. This observation has been used in the branch direction as well as branch target prediction techniques. As an example, consider a code block B with four control transfer instructions I_1, I_2, I_3, I_4. Given a block-grain global history pattern H_1, let us suppose that I_1 and I_2 are predominantly not taken, while

I_3 is taken with the branch target in a different code block that is not the next code block. In this case, a hash of H_1 and the block address of B will be trained with the target of I_3 because that represents the discontinuity in the sequential code stream. On the other hand, let us suppose for a different global history pattern H_2, I_1, I_2, I_3 are predominantly not taken and I_4 is taken with the target in the next sequential code block. In this case, the hash of H_2 and the block address of B will be trained to predict not taken. Next, we discuss the design of the proposed prefetcher.

3.2 Prefetch Target Buffer Design

We incorporate a branch target buffer to carry out the prediction discussed above. We will refer to this structure as the prefetch target buffer (PTB). The PTB is a set-associative array indexed using the XOR of the least significant n bits of the current block address and the most recent n bits of the block-grain global history assuming that the PTB has 2^n sets. Each PTB entry has a valid bit, replacement state bits, a partial tag, a prefetch target, and a 2-bit saturating pattern history counter. The partial tag is same as the least significant m bits of the current block address. In our implementation, n is 11 (the PTB has 2048 sets) and m is 12.

The prefetch target stored in a PTB entry uses a compressed encoding. It has two components, namely the lower target and an upper target pointer. The lower target stores the least significant t bits of the target block address, while the upper target pointer stores a pointer to an entry of a small fully-associative dictionary. The dictionary stores the possible values of upper target bits seen in the recent past. When a new PTB entry is allocated, the dictionary is searched for an entry having the same upper target bits as the prefetch target of the new PTB entry. If found, the corresponding entry id is stored in the upper target pointer of the PTB entry. If not found, a new dictionary entry is allocated using the LRU replacement and the corresponding entry id is stored in the upper target pointer of the PTB entry. The LRU states of the dictionary are updated whenever a dictionary entry is retrieved to construct a prefetch target. This target compression technique is similar to the one used in Seznec's ITTAGE predictor from CBP3 [21]. The ITTAGE design used a static dictionary of 128 entries, while we use a temporally populated dynamic dictionary of much smaller size. In our implementation, the lower target has 14 bits, the upper target pointer has 5 bits, and the dictionary has 32 entries. With 64-byte cache blocks, this arrangement, at any instant of time, can track active prefetch targets over 32 MB of code footprint distributed among 32 1 MB regions. This is usually enough to capture the code locale during a reasonably large execution window. Assuming 64-bit instruction addresses, each dictionary entry needs to store the most significant 44 bits of a prefetch target. Adding five LRU state bits and a valid bit makes a dictionary entry 50-bit wide.

A new entry is allocated in the PTB when a taken block address is discovered in the instruction stream. However, it is observed that a code block may exhibit both taken and not-taken behavior during the execution. To train the PTB with this behavior, we make use of the 2-bit pattern history counter attached to each entry. To update this counter, the PTB is looked up for both taken and not-taken block addresses. If the entry is already present in the PTB, the counter is incremented or decremented using saturating logic for taken or not-taken outcome respectively. If the entry is not present in the PTB, a new entry is allocated only if the outcome is taken and the counter of the newly allocated entry is initialized to 2 (weakly taken). These pattern history counters can be thought of as belonging to a partially tagged gshare predictor (*a la* one component of the TAGE predictor) where a new entry is allocated only when a taken block address is seen. Distributing the gshare counters across the PTB entries saves the space investment of duplicate partial tags.

The replacement policy executed within a PTB set first looks for an invalid entry. If none found, it looks for an entry with pattern history counter value zero (representing strongly not taken behavior in recent past) for replacement. The rationale behind this policy is that ideally the PTB entries should be devoted to only taken block addresses capturing the discontinuities seen in the recent past. If no entry with counter value zero is found within the target PTB set, the static re-reference interval prediction (SRRIP) policy is invoked to find a victim [9]. The SRRIP policy needs only two replacement state bits per PTB entry for storing the re-reference prediction value (RRPV) while offering performance that is never worse than the LRU policy, which requires more replacement state bits per PTB entry for associativity exceeding four.

To generate a prediction, the PTB is indexed using the XOR of the current block address in the instruction stream and the block-grain global history. On a hit, the prediction is taken if the PTB entry's pattern history counter value is at least 2 and in this case, the prefetch target is used for injecting a prefetch; otherwise if the pattern history counter value is less than 2 or there is a PTB miss, the prediction is not taken and a prefetch is injected for the block that is sequentially next to the current block. To look ahead along the control flow path, a copy of the block-grain global history is created (same as checkpointing the global history in branch prediction parlance) and the copy is updated speculatively using the predicted outcome. The XOR of the speculative global history and the predicted block address is used to index into the PTB recursively for generating further prefetch candidates. The process stops when a pre-defined lookahead L is reached. Thus each non-speculative block address in the instruction stream generates L prefetches. The speculative global history is discarded once L prefetches have been generated. The PTB is never updated speculatively. Our implementation uses $L = 11$.

We implement a 28K-entry PTB organized into 2K 14-way sets. Each entry is 36-bit wide. The total size of the PTB is 126 KB. The total size of the dictionary is 32×50 bits or 200 bytes.

3.3 PTB Update and Prefetch Algorithm

The championship infrastructure offers two possible locations in the instruction processing pipeline where the PTB can be updated and looked up for triggering prefetches. One possibility is at the time a demand access looks up the L1 instruction cache. This point in the pipeline offers complete visibility into the demand access stream and one can easily discover the taken and not taken block addresses in the demand stream. Another possibility is right after the branch predictor is looked up. This is before the L1 instruction cache is looked up. This point exposes only the branch instructions and their predicted outcomes. Although this point exposes a filtered instruction stream containing only the branch instructions, this is enough to correctly update the PTB because the left out instruction blocks are part of a basic block and necessarily not taken. Therefore, these blocks are not needed for updating the PTB. We can also maintain a condensed block-grain global history at this point. One advantage of triggering prefetches at the branch prediction point is that when the branch is predicted taken and the prediction is correct, the predicted target immediately offers one step of correct prefetch lookahead. Given the average 99% accuracy of the hashed perceptron branch predictor used in the front-end, we always update the PTB with the predicted target whenever a branch is predicted taken and in these cases, we always trigger a prefetch for the predicted taken target block. Our design updates the PTB and the block-grain global history at the branch prediction point and also triggers prefetches at this point. Our evaluation will show that this is a slightly superior choice compared to triggering prefetches at the time of cache lookup.

We explain our PTB update and prefetching algorithms using two examples depicting two possible cases in the branch instruction stream. Let us consider a branch instruction b_1 belonging to cache block B_1. Following b_1 in the dynamic instruction stream, b_2 is the earliest branch instruction that does not belong to B_1 but belongs to, say, cache block B_2. In the first case, we will consider b_1 to have been predicted taken with the predicted target belonging to cache block B_T. In the second case, we will consider b_1 to have been predicted not taken. In both cases, the last seen code block will be denoted by B_0.

In the first case, when the branch b_1 is encountered, if B_1 is not equal to B_0, but equal to $B_0 + 1$, then B_0 is resolved as not taken and the PTB is updated accordingly. On the other hand, if B_1 is equal to neither B_0 nor $B_0 + 1$, then B_0 is resolved as taken with target B_1 and the PTB is updated accordingly. This is the first update of the PTB when the branch b_1 is encountered. Next, if B_T is not equal to B_1, but equal to $B_1 + 1$, then B_1 is resolved as not taken and the PTB is updated accordingly. On the other hand, if B_T is equal to neither B_1 nor $B_1 + 1$, then B_1 is resolved as taken with target B_T and the PTB is updated accordingly. This is the second update of the PTB. Two PTB updates are needed for the predicted taken branches. At this time, B_0 is assigned the value B_T. When the branch b_2 is encountered, since B_0 has a predicted taken target, it is assumed that B_0 and B_2 are parts of the same basic block and therefore, B_0 is resolved as not taken and the PTB is updated accordingly. All along, the block-grain global history is also updated according to the taken/not-taken outcomes.

The PTB update algorithm in the second case (when b_1 is predicted not taken) is same as the first PTB update in the first case. The second update is not needed in this case. When the branch b_2 is encountered, since B_0 does not have a predicted taken target, the PTB update is similar to the first PTB update when the branch b_1 is encountered i.e., the update is decided by comparing B_2 with B_0 and $B_0 + 1$.

In the first case (when b_1 is predicted taken with target block B_T), the following prefetches would be injected when the branch b_1 is encountered: one for block B_1, one for block B_T, and L (lookahead parameter) prefetches following B_T by recursively looking up the PTB. In the second case (when b_1 is predicted not taken), the following prefetches would be injected when the branch b_1 is encountered: one for block B_1, and L prefetches following B_1. Therefore, depending on the predicted outcome of a branch, $L + 1$ or $L + 2$ prefetches are generated for every encountered branch instruction.

For completeness, we present our PTB update and prefetch algorithm in Algorithm 1. The *current_block* variable represents the code block containing the current branch instruction (B_1 or B_2 in the example above). The *last_block* variable represents the last code block seen by this algorithm (B_0 in the example above). The *current_target* variable is nonzero and holds the predicted branch target if the current branch is predicted taken; otherwise this variable is zero. The *last_target_valid* variable is true if the last seen branch was predicted taken. The $UpdatePTB$ function takes four arguments, namely the block-grain global history (*ghist*), the block address to be used for indexing (along with *ghist*) and tagging, the target block address (marked don't care if not taken), and taken (T)/not-taken (NT). The algorithm assumes that whenever a branch is predicted taken, the prediction is correct, as has been made clear in the example above. This is mostly true given the 99% accuracy of the branch predictor. This is why the PTB and *ghist* are updated based on the predicted taken target whenever a branch is predicted taken. The $LookupPTB$ function takes the global history and a block address to look up the PTB and returns hit/miss, prefetch target, and the pattern history counter value of the looked up PTB entry. The last two outputs are relevant only in the case of a PTB hit. The $prefetch$ function injects a prefetch for the block containing the address passed as the argument to the function. The $lookahead\_prefetch$ function (Algorithm 2) generates a sequence of L prefetches by recursively looking up the PTB and updating the speculative global history (*spec_ghist*).

3.4 Recent Access Filter

Ideally, every prefetch request should be accurate and should generate a new miss from the L1 cache. Unfortunately, it is not easy to figure out which prefetches will hit in the L1 cache without looking up the cache. We approximately answer this question by maintaining a small fully-associative Recent Access buffer for keeping track of the recently seen block addresses in the demand access stream and the recently inserted prefetches. A newly generated prefetch request looks up this buffer and in the case of a hit, the prefetch is dropped. In our implementation, the Recent Access filter has fifteen entries, exercises FIFO insertion, and is organized as a circular FIFO buffer. Each entry stores a 58-bit block address and a valid bit. The total size of the filter is 889 bits including the four-bit write pointer.

3.5 Storage Overhead

The total storage overhead of our design is computed by adding the sizes of the PTB, the dictionary, the Recent Access filter, and auxiliary counters and registers. This total comes to 126.3 KB.

Algorithm 1 PTB update and prefetch algorithm

```
1:  if $last\_block \neq current\_block$ then
2:      if $last\_target\_valid$ then
3:          $UpdatePTB(ghist, last\_block, X, NT)$
4:          $ghist \leftarrow ghist << 1$
5:      else
6:          if $current\_block == last\_block + 1$ then
7:              $UpdatePTB(ghist, last\_block, X, NT)$
8:              $ghist \leftarrow ghist << 1$
9:          else
10:             $UpdatePTB(ghist, last\_block,$
11:                                 $current\_block, T)$
12:             $ghist \leftarrow (ghist << 1)|1$
13: if $current\_target \neq 0$ then
14:     if $current\_target\_block == current\_block + 1$ then
15:         $UpdatePTB(ghist, current\_block, X, NT)$
16:         $ghist \leftarrow ghist << 1$
17:     else
18:         $UpdatePTB(ghist, current\_block,$
19:                                 $current\_target\_block, T)$
20:         $ghist \leftarrow (ghist << 1)|1$
21:     $prefetch(current\_target)$
22:     $ip\_block \leftarrow current\_target\_block$
23:     $spec\_ghist \leftarrow ghist$
24:     $lookahead\_prefetch(ip\_block, spec\_ghist)$
25: if $last\_block \neq current\_block$ then
26:     $prefetch(current\_block << \log(blocksize))$
27:     if $current\_target == 0$ then
28:         $ip\_block \leftarrow current\_block$
29:         $spec\_ghist \leftarrow ghist$
30:         $lookahead\_prefetch(ip\_block, spec\_ghist)$
31: if $current\_target \neq 0$ then
32:     $last\_block \leftarrow current\_target\_block$
33:     $last\_target\_valid \leftarrow true$
34: else
35:     $last\_block \leftarrow current\_block$
36:     $last\_target\_valid \leftarrow false$
```

Algorithm 2 Lookahead prefetching algorithm

```
1:  procedure $lookahead\_prefetch(ip\_block, spec\_ghist)$
2:      $lookahead \leftarrow 0$
3:      while $lookahead < L$ do
4:          $PTBhit \leftarrow LookupPTB(spec\_ghist, ip\_block,$
5:                                 $\&pftarget\_block, \&count)$
6:          if $PTBhit == 0$   OR   $count < 2$ then
7:              $pftarget\_block \leftarrow ip\_block + 1$
8:              $spec\_ghist \leftarrow spec\_ghist << 1$
9:          else
10:             $spec\_ghist \leftarrow (spec\_ghist << 1)|1$
11:         $ip\_block \leftarrow pftarget\_block$
12:         $prefetch(pftarget\_block << \log(blocksize))$
13:         $lookahead \leftarrow lookahead + 1$
```

4. SIMULATION RESULTS

We evaluate our proposal on the ChampSim IPC1 infrastructure configured for single-core evaluation. The infrastructure uses a 32 KB 8-way L1 instruction cache, a 48 KB 12-way L1 data cache, a 512 KB 8-way L2 cache, and a 2 MB 16-way L3 cache. We use the hashed perceptron branch predictor and LRU replacement policy in the L3 cache. The L3 cache has no prefetcher. The L1 instruction cache has no prefetcher in the baseline, while the L1 data cache uses a next-line prefetcher, and the unified L2 cache uses the signature path prefetcher (SPP) [13]. We use 50 single-thread traces captured from client, server, and the SPEC CPU applications. These traces have baseline L1 instruction cache misses per kilo instructions (MPKI) ranging from 4.4 to 81.7 averaging at 35.7.

4.1 Performance Evaluation

Figure 1 shows the speedup achieved by our PTB-based L1 instruction cache prefetcher for each of the fifty traces. Since this prefetcher injects prefetches at the time of branch prediction, we denote it by PTBbr. For comparison, we also evaluate the ideal oracle prefetcher which marks every L1 instruction cache miss as processed immediately after a miss is detected so that to the instruction fetcher every access seems to enjoy a hit. However, the miss request is sent out to the outer levels of the cache hierarchy thereby modeling the bandwidth and cache capacity consumption of the ideal prefetcher which prefetches with zero overfetch and 100% accuracy and coverage. As the figure shows, across the board, PTBbr performs close to the oracle. Only a handful of traces show a noticeable speedup gap between the two. The speedup of PTBbr ranges from 3.2% to 91.3% averaging at 27.6% (GMEAN bar in the bottom panel). On average, the ideal prefetcher achieves a speedup of 30.6%. On one trace of gcc (gcc_3), PTBbr performs better than the ideal prefetcher. We find that this is because with PTBbr, this trace enjoys 6% less LLC miss count compared to the oracle prefetcher leading to lower DRAM congestion in PTBbr. This result points to the fact that it is possible for a prefetcher to optimize other aspects of the cache and memory hierarchy beyond just prefetching ideally in terms of accuracy and coverage. PTBbr, by accident, ends up lowering the DRAM congestion for one trace. However, it is possible to systematically incorporate DRAM bandwidth optimization in the prefetcher.

Figure 2 compares the speedup averaged across the fifty traces for a number of different designs. We discuss each in the following.

Next2L: The next-2-line prefetcher injects prefetches for two sequential blocks following the current code block. We implement a slightly optimized version of it. After the branch prediction completes, this prefetcher injects a prefetch for the block containing the current branch and if the branch is predicted taken, it injects a prefetch for the block containing the predicted target. In addition, if the branch is predicted taken, it injects prefetches for the next two blocks following the block containing the predicted target; if the branch is predicted not taken, it injects prefetches for the next two blocks following the block containing the current branch instruction. The Next2L prefetcher improves performance by 12.5% averaged across fifty traces.

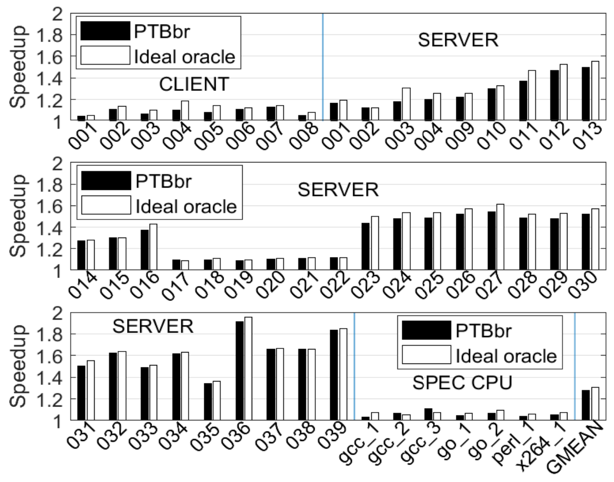

Figure 1: Speedup of fifty traces.

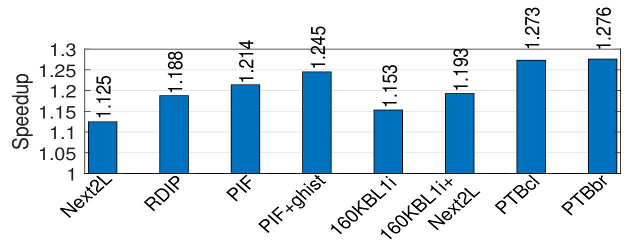

Figure 2: Comparison of average speedup.

RDIP: The return-directed instruction prefetcher (discussed in Section 2) is tuned for the best possible performance within 128 KB storage budget. It is configured to have a 14K-entry region table (1K 14-way sets), which is indexed using the least significant ten bits of XOR of the top four entries in the return address stack (RAS). The next twelve bits of the XOR are used as the tag. Apart from the tag, valid bit, and replacement states, each region table entry contains two region vectors each of size eight bits representing the first two eight-block regions associated with the XOR signature. Each vector is associated to a region address stored in the compressed format using our dictionary. The best lookahead for this prefetcher is found to be four. The prefetcher employs our recent access filter. RDIP improves performance by 18.8% averaged across fifty traces. The primary impediment to performance improvement for RDIP is its limited lookahead. The RAS signature-based region prediction accuracy drops quickly with increasing lookahead.

PIF: The proactive instruction prefetching technique (discussed in Section 2) is tuned for the best possible performance within 128 KB storage budget. It is configured to have a 28K-entry history buffer and an 8K-entry index table (512 16-way sets). Each history buffer entry stores a compressed region address using our dictionary and a bitvector of length eight recording the cache blocks accessed within the eight-block region. Apart from the valid bit and replacement states, each index table entry stores a 13-bit partial tag derived from region address and a 15-bit history buffer pointer. The best lookahead (and degree) for this prefetcher is found to be twelve history buffer entries. The prefetcher employs our recent access filter. PIF improves performance by 21.4% averaged across fifty traces. While prefetching the regions from the history buffer, PIF injects a large volume of unnecessary prefetches because the exact set of blocks in a region does not always repeat.

PIF+ghist: The index table in PIF is looked up using a region address. We design PIF+ghist which looks up the index table using the XOR of the block-grain global history (maintained using our concept of taken/not-taken block addresses) and region address. PIF+ghist improves performance by 24.5% averaged across fifty traces. Combining block-grain global history with region address for looking up the index table offers a sizable improvement in performance over PIF.

160KBL1i: In this design, the entire 128 KB storage budget is invested to design a 160 KB L1 instruction cache (256 10-way sets). The L1 instruction cache has no prefetcher. We also keep the cache access latency unchanged. This design improves performance by 15.3% averaged across fifty traces indicating that a 160 KB L1 instruction cache can bridge only half the performance gap between the baseline 32 KB L1 instruction cache and the ideal oracle prefetcher.

160KBL1i+Next2L: The 160 KB L1 instruction cache is augmented with the Next2L prefetcher. This design improves performance by 19.3% averaged across fifty traces.

PTBcl: In this design, our proposed PTB is updated and looked up for injecting prefetches at the time of L1 instruction cache lookup. PTBcl improves performance by 27.3% averaged across fifty traces.

PTBbr: In this design, our proposed PTB is updated and looked up for injecting prefetches at the time of branch prediction. PTBbr improves performance by 27.6% averaged across fifty traces. The reason for the performance gap between PTBcl and PTBbr was discussed in Section 3.3.

Figure 3 shows, for different designs, the L1 instruction cache miss count averaged over the fifty traces and normalized to the baseline. For each design, the misses are categorized into demand and prefetch misses. On top of each bar, we show the percentage of prefetch misses that ultimately turn out to be useful i.e., the prefetched block is consumed by a demand access before getting evicted. This percentage is an indication of prefetch accuracy and timeliness. We find that PIF+ghist, PTBcl, and PTBbr have the lowest volume of demand misses indicating best prefetch coverage among the designs. They eliminate respectively 84.5%, 85.1%, and 83.7% demand misses relative to the baseline. However, PIF+ghist injects a lot more prefetches than PTBcl and PTBbr to achieve the nearly same coverage. While PIF+ghist sees an 84.8% overfetch, PTBcl and PTBbr overfetch by 33.8% and 43.6% respectively. Additionally, the prefetch accuracy of PIF+ghist is only 63%, while PTBcl and PTBbr achieve an accuracy of 81% and 77% respectively. Overall, PIF+ghist, despite having similar prefetch coverage as PTBcl and PTBbr, falls short in terms of performance due to high overfetch leading to lower prefetch accuracy and higher congestion in the memory hierarchy. Nonetheless, PIF+ghist is successful in lowering the overfetch and increasing the prefetch accuracy significantly compared to PIF. Notably, RDIP sees zero overfetch and nearly perfect (96%) prefetch accuracy. However, it falls short in terms of coverage and this shortcoming arises from its low lookahead. Among PTBcl and PTBbr, the former has better coverage, better accuracy, and lower overfetch (the differences are small though), yet the latter offers better overall performance. This is primarily due to an overall lower average demand miss latency of PTBbr arising from early injection of prefetches to the blocks containing the predicted taken branch targets.

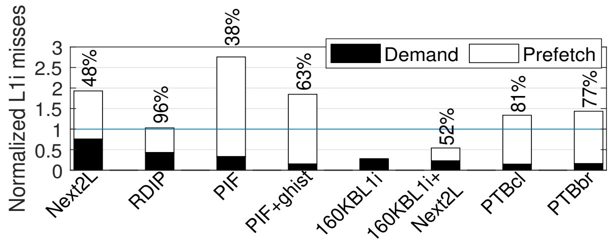

Figure 3: Normalized L1 instruction cache miss count.

Figure 4 shows the L1 instruction cache hit rates for different designs. PIF+ghist, PTBcl, and PTBbr achieve the highest hit rates. Compared to the 81.3% hit rate of the baseline, these three designs achieve hit rates of 98.3%, 98.0%, and 97.7% respectively.

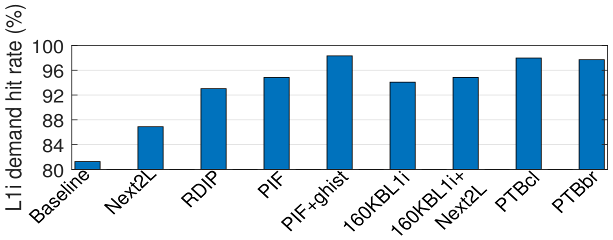

Figure 4: L1 instruction cache hit rate.

4.2 Analysis of the Prefetches

In this section, we further analyze the prefetches injected by the PTBbr design. As discussed in Algorithms 1, the PTBbr design first injects one or two prefetches depending on whether the current branch is predicted not taken or predicted taken, and then, as shown in Algorithm 2, it injects L prefetches that we will refer to as the lookahead prefetches. We configure our design to have $L = 11$. We will refer to the initial prefetches injected by Algorithms 1 as depth-zero (d0) prefetches. The subsequent L prefetches injected by Algorithm 2 will have monotonically increasing depth and will be referred to as d1 to d11 prefetches. Figure 5 shows the distribution of injected prefetches across different lookahead depths.

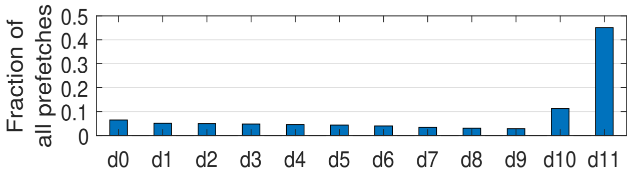

Figure 5: Lookahead depth-wise prefetch distribution.

The d11 prefetches contribute 45% of all prefetches, while d10 prefetches contribute 11%. The contribution of each of d0 to d9 prefetches hovers between 2% and 7%. The reason for this skewed distribution is that the d0 to d10 prefetches injected by the current branch instruction would match the d1 to d11 prefetches injected by the previous branch instruction under an ideal scenario where our PTB offers 100% accurate speculation regarding the blocks touched along the control flow path. In such a situation, our recent access filter would not inject the d0 to d10 prefetches generated by the current branch instruction. Since the PTB-based speculation is not 100% accurate, we see occasional contributions from d0 to d10 lookahead depths that are far below the contribution of the frontline d11 prefetches which essentially push the prefetch wavefront forward. These data confirm that the prediction accuracy of the PTB is quite high and the small recent access filter is quite effective in removing duplicate prefetches. This analysis also brings out the lower bound L on the size of the recent access filter. Our configuration uses four extra entries in the filter to accommodate demand accesses i.e., our recent access filter has $L+4$ entries.

An injected prefetch can meet four possible fates: (i) it can hit in the L1 instruction cache, (ii) it can miss in the L1 instruction cache and prefetch the block before demanded, (iii) it can miss in the L1 instruction cache and get demanded before the prefetch completes, and (iv) it can find an already outstanding demand miss to the block being prefetched. In the following, we analyze the first two cases. In the next section, we will analyze the remaining two cases. Figure 6 shows the distribution of prefetch hits across prefetch depths. Given that most prefetches come from d11 group, it is not surprising that most prefetch hits are sourced by this group of prefetches. The contributions of d0 to d10 are under 10%. It may seem tempting to design a filter that can eliminate all prefetch hits because prefetch hits apparently do not contribute anything to the end-performance. However, we find that the replacement state update done by a prefetch hit sometimes helps retain the accessed block longer in the cache. This is important because a prefetch, if accurate, indicates an imminent demand access to the same block. In fact, disabling replacement state update of all prefetch hits leads to a small drop in average speedup from 27.6% to 27.4% in PTBbr.

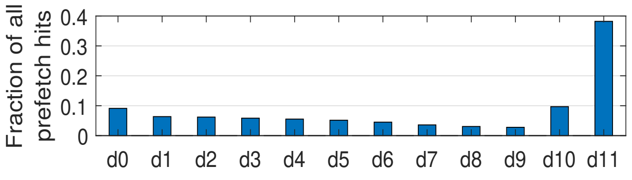

Figure 6: Lookahead depth-wise prefetch hit distribution.

Figure 7 shows the distribution of L1 instruction cache prefetch misses across lookahead depths. For each lookahead depth, we show its contribution toward prefetch misses divided into two categories, namely useful prefetch misses and useless prefetch misses. Overall, 77% prefetch misses are useful and 23% are useless. Most of the useful prefetch misses (60% of total 77%) are contributed by d10 and d11. Unfortunately, most of the useless prefetch misses are also contributed by d10 and d11 (16% of total 23%). As a result, a prefetch depth-based filtering for useless prefetches won't work. The prefetch depths d0 to d9 contribute very little (each under 2%) to useful prefetches. It may be tempting to conclude that some of these prefetch depths may be eliminated altogether. To quantify the relative importance of each prefetch depth, we evaluate PTBbr by gradually disabling prefetch injection from depths starting from d0. For example, PTBbr evaluated with d6-d11 prefetch depths injects prefetches only for depths d6 to d11 and does not inject any prefetch for depths d0 to d5. Figure 8 shows these speedup numbers. These data show that the speedup gradually declines as prefetch injection from a bigger block of prefetch depths is disabled indicating that each prefetch depth contributes positively to the end-performance. A filter for useless prefetches would need to be more selective even within each prefetch depth.

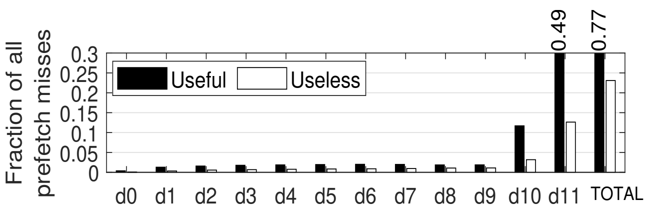

Figure 7: Lookahead depth-wise prefetch miss distribution.

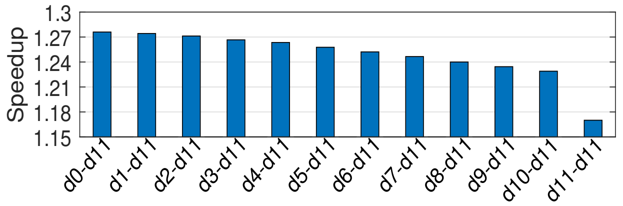

Figure 8: Speedup with prefetch injection enabled for a subset of prefetch depths. The enabled subsets are enumerated on the x-axis as a range of enabled prefetch depths.

The useless prefetches create congestion throughout the memory hierarchy and pollute the cache levels wherever they are filled. This may further increase the number of misses at a cache level because a useless prefetch may replace a useful block prematurely. Figure 9 shows the number of misses (demand and prefetch taken together) and the average miss latency at different cache levels normalized to the baseline. Our proposal increases the miss count of the L1 instruction cache by 43.6%, but decreases the average miss latency by 3.2%. The miss count of the L1 data cache remains unaffected, as expected, while the average miss latency increases by 3.9% due to the congestion created in the memory hierarchy by the useless code prefetches. The miss counts of the L2 cache and the LLC increase by 22.9% and 8.9% respectively, while their average miss latency numbers increase by 6.9% and 7.2% respectively. Overall, even though PTBbr increases the total demand plus prefetch miss counts at the L1 instruction cache and the L2 cache quite significantly, the increase in the average miss latency is under 8% at all cache levels. Our performance results show that this increase in the average miss latency at the L1 data cache, L2 cache, and LLC is more than compensated by a reasonably large drop in the L1 instruction cache demand miss count coupled with a small drop in the average L1 instruction cache miss latency. This latency drop is explained in the next section.

4.3 Analysis of the Demand Misses

As already discussed, the PTBbr prefetcher is able to convert 83.7% demand misses into hits for the L1 instruction cache. In this section, we characterize the residual 16.3%

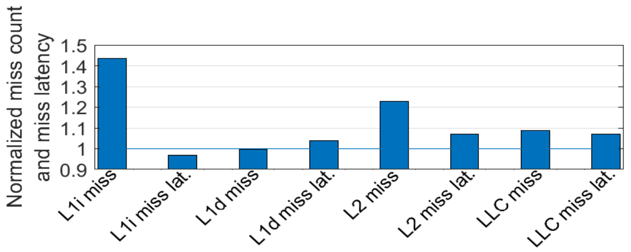

Figure 9: Normalized miss count and miss latency at different levels of cache hierarchy.

demand misses. A demand miss can belong to two categories: (i) a demand access that misses in the cache, and (ii) a demand access that finds an already outstanding prefetch miss to the same block. The demand misses in the second category are sometimes referred to as partial hits because in these cases a fraction of the miss latency gets hidden due to an already initiated prefetch. The demand misses in the first category have an important subcategory as far as prefetches are concerned: (i-a) a demand miss that receives a prefetch to the same block while it is outstanding. These prefetches are quite late and are dropped because they are of no use, but it is helpful to understand how many such cases arise.

Figure 10 characterizes the demand misses in categories (i-a) and (ii) for the PTBbr prefetcher. Depending on the depth of the prefetch that merges with the demand miss, the two categories are further divided into the prefetch depth bins. Overall, 82% of the demand misses belong to category (ii) i.e., partial hits (the left bar in each group) and only 7% belong to category (i-a). Therefore, 89% of the demand misses have corresponding late prefetches indicating fairly high "partial coverage" and accuracy of the prefetcher. Only 11% of the demand misses (which is 11% of 16.3% i.e., 2% of baseline demand misses) cannot be covered by the prefetcher. The biggest contributor to the partial hits are the prefetches from depth d1 (contributes 20% out of 82% i.e., nearly one-quarter), while depths d0, d2, and d11 each contributes more than 10%. These data confirm that most partial hits arise from prefetches with low lookahead depths (depths d0 to d4 together contribute 57% out of 82%) and this is an expected behavior because the late prefetches are most likely to come from low lookahead depths. The category (i-a) demand misses also primarily arise from prefetch depths d0 to d4, but these are much smaller in volume, which is encouraging because we do not want to have such late prefetches in big numbers.

An important parameter that characterizes the partial hits is the fraction of the average miss latency that gets hidden in these cases. We find that 43% of the miss latency gets hidden in the partial hit cases meaning that only 57% of the miss latency of the prefetch gets exposed on the critical path of the demand miss that merges with the outstanding prefetch. This large saving in the average miss latency of 82% of the demand misses leads to an overall drop in the average L1 instruction cache miss latency as was pointed out in Figure 9 of the last section. In summary, the data in Figure 10 indicate that one possible way to further improve performance is by improving the timeliness of the prefetches injected by the lower lookahead depths. Additionally, the prefetch coverage needs to be improved to capture all residual demand misses.

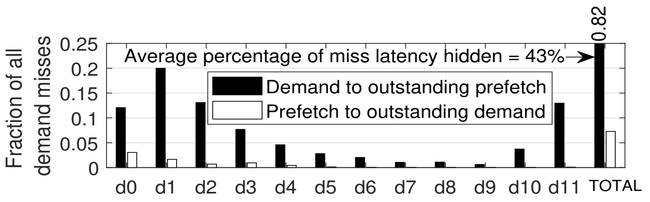

Figure 10: Characterizing demand misses that merge with prefetches.

4.4 Bridging the Residual Performance Gap

The PTBbr prefetcher leaves a speedup gap of about 3% from the ideal oracle prefetcher when measured relative to the baseline. In this section, we attempt to understand any obvious bottlenecks that could be addressed for narrowing this gap. Figure 11 explores the performance of various idealized configurations. The leftmost two bars present the speedup of our PTBbr proposal and the ideal oracle prefetcher. The third bar from the left shows that if all residual L1 instruction cache demand misses in PTBbr are marked processed immediately after the miss is detected while letting the cache miss fill request proceed as usual, the achievable speedup is 29.8%. This speedup is slightly lower than the original 30.6% of the ideal oracle prefetcher due to the additional congestion/pollution created by the PTBbr prefetcher in the memory hierarchy. The fourth bar shows that if the demand misses resulting in partial hits (i.e., a prefetch miss is already outstanding) are marked processed immediately after they merge with the outstanding prefetch (phit-oracle), the achievable speedup is only 28.3%. Therefore, just improving the timeliness of the partial hits would not be enough even though these partial hits account for 82% of the residual L1 instruction cache demand misses in PTBbr. Improving these partial hits has low return because 43% of the average latency of these misses is already saved. The remaining 18% of the residual demand misses need to be covered with prefetches.

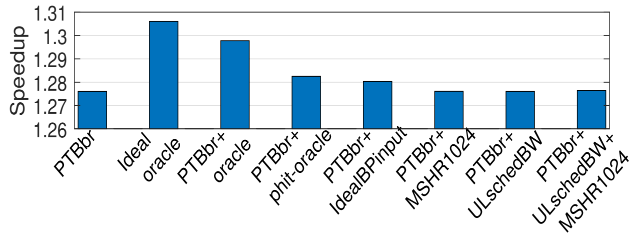

Figure 11: Performance speedup for different idealized configurations.

PTBbr relies on the accuracy of the hashed perceptron branch predictor in the predicted taken cases because in these cases, the starting point of the chain of L lookahead prefetches is the predicted taken branch target. The PTBbr+IdealBPinput bar examines the achievable speedup if always correct branch predictions are offered to PTBbr. In this case, the speedup increases to only 28%. The speedup improvement compared to 27.6% of PTBbr is rather small due to already high (99% on average) prediction accuracy of the hashed perceptron branch predictor.

The PTBbr+MSHR1024 bar shows that the speedup does not improve at all even if the number of L1 instruction cache MSHRs is made 1024 indicating that MSHR is not a bottleneck. The PTBbr+ULschedBW evaluates the speedup when the prefetch scheduling bandwidth at the L1 instruction cache is made unlimited meaning that all pending prefetches in the prefetch queue can be issued in the same cycle. In this case also, the speedup does not improve indicating that prefetch scheduling bandwidth is not a bottleneck. The rightmost bar offers 1024 MSHRs to PTBbr+ULschedBW, but still cannot improve the speedup. Overall, the timeliness of the prefetches cannot be improved by equipping the front-end with more MSHR or prefetch scheduling resources.

In summary, discovering new correlations to improve the accuracy and coverage of the PTBbr prefetcher seems to hold the key to bridging the residual performance gap between PTBbr and the oracle. One possible approach in this direction could be to synthesize the code prefetcher from more sophisticated direction and target prediction mechanisms that the branch predictors have employed. Examples include hashed perceptron and TAGE predictor for predicting the direction and ITTAGE and bit-level perceptron prediction for predicting the target [7].

Before closing this discussion, we examine the efficiency of content management of the PTB. On average, only 17% of the allocated PTB entries are not used before getting evicted. Therefore, we do not expect a significant advantage to come from optimization of the PTB management algorithms. Nonetheless, to verify this hypothesis, Figure 12 evaluates the performance of PTBbr as the number of PTB entries is increased up to 8M. The achievable speedup saturates at around 28.3%. This presents a small potential benefit on top of the PTBbr speedup of 27.6% exercising a 28K-entry PTB.

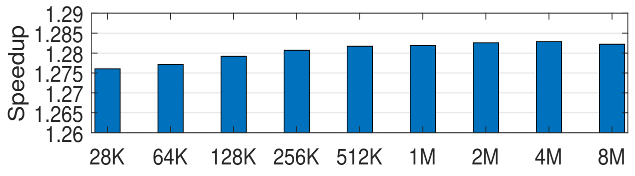

Figure 12: Speedup variation in PTBbr with increasing PTB entries.

5. PTB MICROARCHITECTURE

As discussed in Section 3.3, one branch instruction may trigger up to two PTB writes and $L+2$ PTB reads. Since the two writes are completely independent, they can be issued in parallel provided the PTB has two write ports. However, two writes are needed only for the predicted taken branches. So, provisioning the PTB with two write ports would be a wastage of resources and that would also slow down the PTB. We design the PTB as an eight-way banked structure where the PTB sets are distributed across the banks round-robin i.e., the lower three bits of the PTB index constitute the bank number. Each PTB bank is provisioned with a single write port. As a result, if two writes go to two different banks, they can be done in parallel. However, there remains a possibility of bank conflict between the writes. To buffer such conflicted writes, each PTB bank maintains a four-entry circular FIFO write buffer. Each buffer entry contains enough information to complete the write at a later time (PTB index within the bank, PTB tag, prefetch target, taken/not taken, valid bit). Each write buffer also has a write pointer and a read pointer to respectively enqueue and dequeue entries. The total size of the write buffer including the read/write pointers aggregated over eight PTB banks is 324 bytes. Each cycle the write at the head of each PTB bank's write buffer is scheduled provided the buffer is not empty.

Among the PTB reads needed for injecting prefetches, the last $L-1$ reads generated by the *lookahead_prefetch* procedure (Algorithm 2) form an inherently sequential chain because each read in this chain depends on the speculative global history and the prefetch target updated based on the previous read's outcome. This can lead to a drastic drop in the prefetch injection rate. To maintain high prefetch injection rate, we observe that the chains of PTB reads generated by two different branch instructions are completely independent. Therefore, if we consider N different branch instructions, we can execute N PTB reads in parallel and inject as many prefetches provided the PTB has enough read ports. To implement this idea, we maintain N read scheduling queues. Each queue is implemented as a circular FIFO buffer. Each entry of a queue is populated by a new source/trigger instruction (either a branch instruction or a predicted taken target) of a prefetch and the entry holds the *spec_ghist*, *ip_block*, and *lookahead* values from Algorithm 2. Each queue maintains write and read pointers for enqueuing and dequeuing purpose. Consecutive source/trigger instructions populate the tail entries pointed to by the write pointers of consecutive queues in a round-robin fashion with the help of a queue pointer so that the population across the queues remains balanced.

Every cycle the head entry of each read scheduling queue is picked up and one iteration of the `while` loop from Algorithm 2 is executed for that entry subject to availability of PTB read ports. After execution, the entry's *spec_ghist*, *ip_block*, and *lookahead* values are updated as is done in each iteration of the `while` loop in Algorithm 2. If the *lookahead* value of the head entry of a queue has reached L, the read pointer of that queue is advanced. We find that the best performance is obtained when N, the number of queues, matches L so that a peak prefetch injection rate of L per cycle is maintained. We provision each PTB bank with four read ports. For $L = 11$ (as per our configuration), ideally 11 read ports would be needed in the PTB. However, to avoid port conflicts in a bank, the banks are over-provisioned with read ports. The PTB organization is shown in Figure 13. Each of the 11 circular FIFO queues is sized so that the overall storage overhead remains within 128 KB, the budget provisioned by the championship. Each queue has 13 entries and the total storage needed for these queues including the read, write, and queue pointers is 1.29 KB. Taken together, the overhead of the per-bank write buffers, the read scheduling FIFO queues, the PTB port occupancy bitmaps, and per-cycle prefetch injection states is 1.61 KB. The previously computed total overhead was 126.3 KB. Thus, the total overhead after including the PTB microarchitecture details is 127.91 KB.

We measure the access latency of one PTB bank (256 14-way sets) having four read ports and one write port using CACTI [8]. To make the analysis amenable to CACTI, we slightly oversize each PTB bank to have 256 sets, 16 ways, 4-byte blocks, and 17-bit tags. For 22 nm technology, CACTI reports an access latency of 0.45 ns meeting the cycle time of a 2 GHz clock. The per-bank area reported by CACTI is 0.18 mm^2. For a front-end with higher than 2 GHz frequency, the PTB access needs to be pipelined to avoid losing performance. This implementation offers an average speedup of 27.3% over the baseline. This is close to the speedup of 27.6% achieved by the unconstrained design evaluated in the last section.

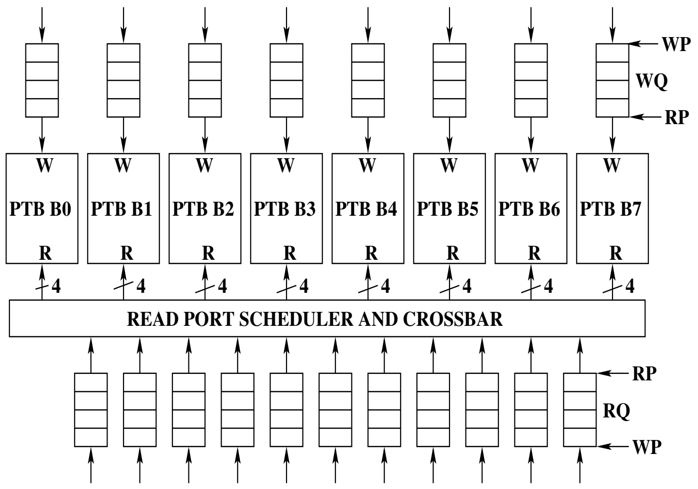

Figure 13: PTB microarchitecture. RQ and WQ are respectively read scheduling and write queues. RP and WP are respectively read and write pointers. Within a PTB bank, the write port is denoted by W and the read ports are denoted by R.

6. SUMMARY

We have presented the design of an L1 instruction cache prefetcher that directly inherits its operations and storage structures from well-known control speculation hardware. This is made possible by posing the code prefetching problem as a control flow speculation problem at the grain of cache blocks. The proposed prefetcher sports a branch target buffer-like structure referred to as the prefetch target buffer. Each prefetch target buffer entry also embeds a pattern history counter derived from a partially tagged gshare predictor. Compared to a baseline that has no L1 instruction cache prefetcher, the proposed prefetcher achieves a speedup of 27.6% averaged over fifty dynamic instruction traces collected from client, server, and SPEC CPU applications. A more realistic implementation that takes into account port constraints of the prefetch target buffer and latency constraints of prefetch injection achieves an average speedup of 27.3%. This performance comes close to the ideal oracle prefetcher which speeds up execution by 30.6% on average while offering 100% L1 instruction cache hit rate.[1]

[1] Source codes available at https://www.cse.iitk.ac.in/users/mainakc/code_prefetchers.html.

7. REFERENCES

[1] I. Burcea and A. Moshovos. Phantom-BTB: A Virtualized Branch Target Buffer Design. In *ASPLOS* 2009, pages 313–324.

[2] G. Chadha, S. Mahlke, and S. Narayanasamy. EFetch: Optimizing Instruction Fetch for Event-driven Web Applications. In *PACT* 2014, pages 75–86.

[3] I-C. K. Chen, C-C. Lee, and T. N. Mudge. Instruction Prefetching Using Branch Prediction Information. In *ICCD* 1997, pages 593–601.

[4] M. U. Farooq, L. Chen, and L. K. John. Value-based BTB Indexing for Indirect Jump Prediction. In *HPCA* 2010.

[5] M. Ferdman, C. Kaynak, and B. Falsafi. Proactive Instruction Fetch. In *MICRO* 2011, pages 152–162.

[6] M. Ferdman, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Temporal Instruction Fetch Streaming. In *MICRO* 2008, pages 1–10.

[7] E. Garza, S. Mirbagher-Ajorpaz, T. A. Khan, and D. A. Jimenez. Bit-level Perceptron Prediction for Indirect Branches. In *ISCA* 2019, pages 27–38.

[8] HP Labs. CACTI: An Integrated Cache and Memory Access Time, Cycle Time, Area, Leakage, and Dynamic Power Model. Available at http://www.hpl.hp.com/research/cacti/.

[9] A. Jaleel, K. B. Theobald, S. C. Steely Jr., and J. Emer. High Performance Cache Replacement using Re-reference Interval Prediction (RRIP). In *ISCA* 2010, pages 60–71.

[10] C. Kaynak, B. Grot, and B. Falsafi. SHIFT: Shared History Instruction Fetch for Lean-core Server Processors. In *MICRO* 2013, pages 272–283.

[11] C. Kaynak, B. Grot, and B. Falsafi. Confluence: Unified Instruction Supply for Scale-out Servers. In *MICRO* 2015, pages 166–177.

[12] H. Kim, J. A. Joao, O. Mutlu, C. J. Lee, Y. N. Patt, and R. Cohn. VPC Prediction: Reducing the Cost of Indirect Branches via Hardware-based Dynamic Devirtualization. In *ISCA* 2007, pages 424–435.

[13] J. Kim, S. H. Pugsley, P. V. Gratz, A. L. N. Reddy, C. Wilkerson, and Z. Chishti. Path Confidence based Lookahead Prefetching. In *MICRO* 2016.

[14] A. Kolli, A. Saidi, and T. F. Wenisch. RDIP: Return-address-stack Directed Instruction Prefetching. In *MICRO* 2013, pages 260–271.

[15] R. Kumar, C-C. Huang, B. Grot, and V. Nagarajan. Boomerang: A Metadata-free Architecture for Control Flow Delivery. In *HPCA* 2017, pages 493–504.

[16] R. Kumar, B. Grot, and V. Nagarajan. Blasting through the Front-end Bottleneck with Shotgun. In *ASPLOS* 2018, pages 30–42.

[17] S. Mirbagher-Ajorpaz, E. Garza, S. Jindal, and D. A. Jimenez. Exploring Predictive Replacement Policies for Instruction Cache and Branch Target Buffer. In *ISCA* 2018, pages 519–532.

[18] A. Perais, R. Sheikh, L. Yen, M. McIlvaine, and R. D. Clancy. Elastic Instruction Fetching. In *HPCA* 2019, pages 478–490.

[19] G. Reinman, B. Calder, and T. Austin. Fetch Directed Instruction Prefetching. In *MICRO* 1999, pages 16–27.

[20] A. Seznec and P. Michaud. A Case for (Partially) Tagged Geometric History Length Branch Prediction. In *JILP*, vol. 8, 2006.

[21] A. Seznec. A 64-Kbytes ITTAGE Indirect Branch Predictor. In *CBP3* 2011.

[22] A. J. Smith. Sequential Program Prefetching in Memory Hierarchies. In *IEEE Computer*, **11**(12): 7–21, 1978.

[23] A. J. Smith. Cache Memories. In *ACM Computing Surveys*, **14**(3): 473–530, September 1982.

[24] J. E. Smith and W.-C. Hsu. Prefetching in Supercomputer Instruction Caches. In *ICS* 1992, pages 588–597.

[25] L. Spracklen, Y. Chou, and S. G. Abraham. Effective Instruction Prefetching in Chip Multiprocessors for Modern Commercial Applications. In *HPCA* 2005, pages 225–236.

[26] V. Srinivasan, E. Davidson, G. Tyson, M. Charney, and T. Puzak. Branch History Guided Instruction Prefetching. In *HPCA* 2001, pages 291–300.