Enhancing Signature Path Prefetching with Perceptron Prefetch Filtering

Eshan Bhatia[1], Gino Chacon[1], Elvira Teran[2], Paul V. Gratz[1] and Daniel A. Jiménez[3]

[1]Texas A&M University, {eshanbhatia22, ginochacon, pgratz}@tamu.edu

[2]Texas A&M International University, elvira.teran@tamiu.edu

[3]Texas A&M University and Barcelona Supercomputing Center, djimenez@tamu.edu

ABSTRACT

In this paper, we evaluate an implementation of a multi-cache-level prefetching system under the 3rd Data Prefetching Championship framework. Our approach is based on enhancing a baseline prefetcher, Signature Path Prefetching (SPP) with a new, perceptron-based scheme of prefetch filtering. The key idea here being that by de-constraining/de-throttling the underlying prefetcher we can achieve high coverage, while using the perceptron prefetch filter to ensure high accuracy.

Our evaluation on the memory intensive traces of SPEC CPU 2017 suite shows that this approach enhances the IPC of the system by 40.4% over no prefetching for a single-core configuration and by 20.3% for a four-core configuration. On the system, the winner of DPC-2 achieves the resultant speedup of 28.4% and 15.1% in single-core and four-core systems respectively.

1. INTRODUCTION

Prefetchers are designed around a fundamental trade-off between two important metrics: coverage and accuracy. Prefetcher coverage refers to the fraction of baseline cache misses that the prefetcher brings to the cache before their reference. Accuracy refers to the fraction of prefetched cache lines that are actually used by the application. The key idea in this paper is to efficiently balance this trade-off. Our approach yields a prefetching mechanism that can learn complex pointer-chasing patterns (high coverage) and yet work well on constrained multi-core systems (high accuracy).

A key challenge in prefetching for multi-level cache hierarchies lies in designing a coordinated prefetch design approach. This involves controlling the inter-hierarchy prefetch communication like prefetch misses from L1D appearing as accesses to L2C. Another aspect to consider is placement of the incoming prefetches. A correct prefetch suggestion placed in the wrong level, say L1D, can potentially do more harm by wasting precious resources.

In this paper, we follow a modular approach of explaining our basic prefetcher – SPP, our perceptron prefetch filtering scheme – PPF, and the further enhancements included to address the trade-offs described above. Finally, we discuss fitting together the pieces across the cache levels in the final prefetching mechanism implemented.

In a single core configuration, running a mix of memory intensive SPEC CPU 2017 traces, our prefetcher increases performance by 40.4% compared to no prefetching. In a four-core system, it saw an improvement of 20.3% over the baseline.

2. BACKGROUND

In this section we discuss existing approaches that have been used in our submission, namely: Signature Path Prefetcher and Perceptron-based on-line learning.

2.1 Underlying prefetcher

The original Signature Path Prefetcher [1] was proposed by Kim *et. al.* SPP is a confidence-based lookahead prefetcher. It creates a signature associated with a page address by compressing the history of accesses. By correlating the signature with future likely delta patterns, SPP learns both simple and complex memory access patterns quickly. By pushing the predicted delta on to the existing signature, a new speculative signature can be generated which allows further prefetching down a speculative path within the page.

Signature Table: The Signature Table is indexed using the page number and captures memory access patterns within a page boundary. It does so by storing the last few memory accesses in the form of a compressed 12-bit signature, given as:

$$NewSignature = (OldSignature << 3bits)\ XOR\ (Delta)$$

Delta is the numerical difference between the block offset of the current and the previous memory access. This signature is used to index into the Pattern Table. This process is illustrated in Figure 1.

Pattern Table: The Pattern Table, shown on the right side in Figure 1 is indexed by the signature generated from the Signature Table. The Pattern Table holds predicted delta patterns and their confidence estimates. Each entry is indexed by the signature and holds up to 4 unique delta predictions.

Global History Register: The Global History Register tries

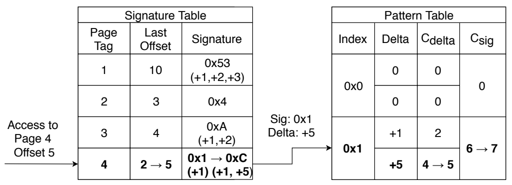

Figure 1: SPP Data-path Flow

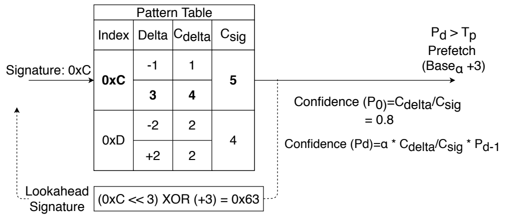
Pattern Table
Index
Delta
C_delta
C_sig
Signature: 0xC
0xC
-1
1
5
3
4
0xD
-2
2
4
+2
2
P_d > T_p Prefetch (Base_α +3)
Confidence (P_0)=C_delta/C_sig = 0.8
Confidence (Pd)=α * C_delta/C_sig * P_d-1
Lookahead Signature
(0xC << 3) XOR (+3) = 0x63

Figure 2: SPP Architecture

to learn from the prefetch suggestions that were rejected as they crossed page boundaries. By bootstrapping its learning from such prefetch suggestions, SPP learns about page transitions, enabling SPP to have a quicker warm-up period for unseen pages.

Lookahead Prefetching: On each trigger, SPP uses its own prefetch suggestion to iteratively generate new prefetch suggestions. Using the current prefetch as a starting point, it re-accesses the Pattern Table to generate further prefetches. As illustrated in Figure 2, it repeats the cycle of accessing Pattern Table and updating the signature based on highest confidence prefetch from the last iteration. The iteration counter on which SPP manages to predict prefetch entries in the lookahead manner is characterized as its 'depth'. While doing so, SPP also continues compounding the confidence at each step such that as depth increases, overall confidence decreases.

Confidence Tracking: SPP scores its prefetch suggestions, denoted by C_d. The score is approximated as the ratio of the hits for a given delta per signature to the hits for that signature. In the lookahead mode, the path confidence P_d is given as:

$$P_d = \alpha \,.\, C_d \,.\, P_{d-1}$$

Here α represents the global accuracy, calculated as the ratio of the number of prefetches which led to a demand hit to the number of prefetches recommended in total. The range of α is [0,1]. The lookahead depth is represented by d. The final P_d is thresholded against pre-defined thresholds to decide whether to prefetch or not and to decide the fill level.

2.2 Perceptron Learning

Perceptron learning for microarchitectural prediction was originally introduced for branch prediction [2]. Our predictor uses a version of a microarchitectural perceptron prediction known as the "hashed perceptron" [3]. The hashed perceptron predictor hashes several different features into values

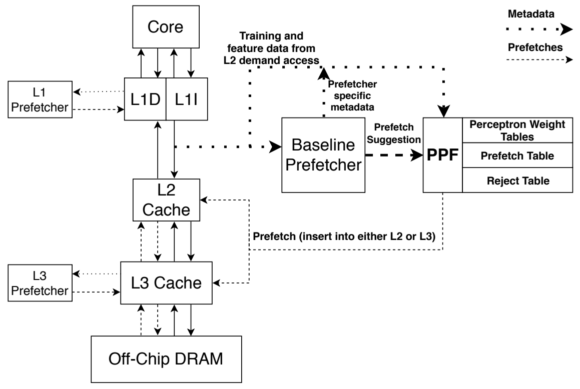

Figure 3: PPF Architecture in the Memory Hierarchy

that index several distinct tables. Small integer weights are read out from the tables and summed. If the sum exceeds a certain threshold, a positive prediction is made, *e.g.* "predict branch taken" or "allow the prefetch." Otherwise, a negative prediction is made. Once the ground truth is known, the weights corresponding to the prediction are incremented if the outcome was positive, or decremented if it was negative. This update only occurs if the prediction was incorrect or if the magnitude of the sum failed to exceed a threshold. Beyond branch prediction, perceptron learning has been applied to last-level cache reuse prediction [4, 5]. In this paper, we apply it to prefetch filtering.

3. PREFETCHING ENHANCEMENTS

It is beneficial to allow a prefetcher like SPP to speculate as deeply as possible. Often, some useful prefetches are generated long after the confidence of the prefetcher has fallen below the point at which prefetcher inaccuracy would lead to performance degradation (*i.e.* coverage continues increasing far beyond the point at which accuracy drops). In order to allow deep speculation in the prefetcher, inaccurate prefetches must be filtered out. We propose to leverage perceptron-based learning as a mechanism to differentiate between potentially useful deeply speculated prefetches and likely not-useful ones. The Perceptron based Prefetch Filter (PPF) is placed between the prefetcher and the prefetch insertion queue, as illustrated in Figure 3, to prevent not-useful prefetches from polluting the higher levels of the memory hierarchy. Perceptron filter considers a number of features corresponding to a prefetch, such as the speculation depth, page address and offset, and uses this information as the inputs to our perceptron-based filter in order to predict the usefulness of a prefetch.

3.1 Changes made to SPP

We modify the baseline SPP design so as to de-throttle it, enabling higher coverage, the following changes were made:

Adjusting Aggressiveness Level: The aggressiveness level was adjusted by tuning down the internal throttling mechanism to extreme values to allow all the prefetches to pass through. Thus, the internal confidence mechanism is no longer used to make prefetch or fill-level decisions.

Adding Prefetch and Reject Filters: In our proposed approach, we need to reindex the perceptron tables (explained in section 3.2) for training when the feedback from the prefetch is available. To do that, we added two 1024-entry filters that hold the data required to index these tables. Depending on the decision of the perceptron (prefetch vs reject), the data is stored in the respective filters.

Exporting data between SPP and Perceptron: Perceptron learning uses the metadata associated with a prefetch suggestion as the perceptron features. Some of the features we developed use information derived directly from program execution. Beyond that, SPP also conveys useful information like lookahead depth, signature, and the confidence counter to the perceptron filter.

3.2 The Perceptron Filter

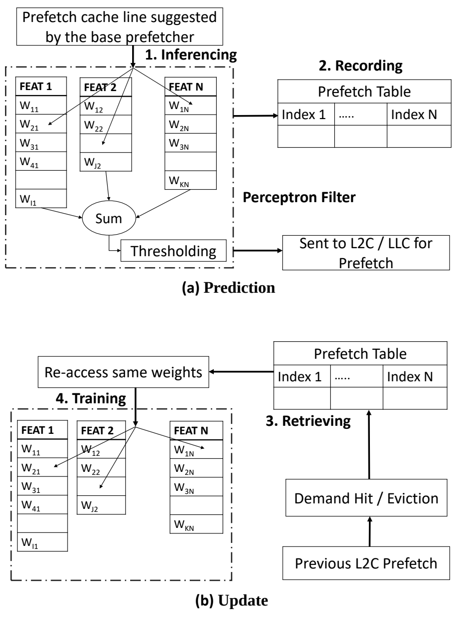

Figure 4: PPF Data Path and Operation

Figure 4 shows the microarchitecture of PPF, as well as the steps required to filter out not-useful prefetches. The perceptron filter is organized as a set of tables, where each entry in the tables holds a *weight*. Each table is indexed using a different number of bits from the corresponding feature. Each weight is a 5-bit saturating counter ranging from -16 to +15.

Inferencing and Data Recording: SPP is triggered on every demand access to the L2 Cache. The suggested prefetch candidates from SPP are fed to the perceptron filter to determine their usefulness and fill-level. This is done by thresholding the perceptron dot-product sum against two different values: τ_{hi} and τ_{lo}. Depending on the outcome of the perceptron, the feature values are recorded in one of the two filters introduced above.

Data Retrieval and Training: PPF training is triggered whenever a prefetched block leads to a demand hit or a cache eviction. A valid entry in one of the tables means that the current memory access was a prefetch candidate in the past. The retrieved feature values are used to index the tables of weights. A uniform perceptron update rule is followed. If the prediction was correct, *i.e.* if a prefetched cache line led to a demand hit, and the perceptron sum lies within a predefined threshold, weights are updated in the correct direction. If the prediction was incorrect, *i.e.* a prefetched cache line was evicted without being used or a rejected prefetch led to a demand access, the weights are updated in the opposite direction.

Note: More details about PPF working and implementation can be found in the paper Perceptron-based Prefetch Filtering [6].

3.3 Other Optimizations

Resource Division Across Pages: For prefetch friendly applications, an aggressive lookahead prefetcher can go deep down the speculation path. This process takes up valuable resources in the system's Prefetch Queue (PQ), blocking any prefetch attempts from subsequent pages and leading to a timing disparity between demand accesses interleaved across pages. To avoid that, our prefetcher maintains a count of distinct pages accessed in the last eight demand accesses and divides the PQ resources across those many pages by limiting the maximum number of prefetches in a given page.

4. PREFETCHER CONFIGURATIONS

This section describes the approach in putting together all the prefetching components across the cache hierarchies. An overview of various prefetching components with respect to the cache hierarchy can be seen in Figure 3

4.1 Single-Core Configuration

1st Level Data Cache: For the L1D Cache we are using a modified version of the next-N-line prefetcher [7]. The basic next line prefetcher is modified to have a small table containing the last block accessed by a page. The table is indexed by hashing the page number of an access. When an access occurs, the current block access and the previous access are compared. If the delta between accesses is +1, then a score table is indexed by the page number and its value increased. If the delta is not +1, it is decreased. When prefetching, the score table is accessed and if the value is above a specified threshold, the next cache line after the access is prefetched. This throttling allows for the prefetcher to be aware if the page is susceptible to multiple +1 deltas, usually consecutively. If the page does not benefit from next line prefetching, the prefetcher is turned off so that it does not risk polluting the L1D and wrongfully evicting data that is more beneficial to performance.

The prefetcher continues to prefetch the next N consecutive lines of that page. N is obtained dynamically by sharing the Prefetch Queue (PQ) resources over consecutively active pages, as explained in Section 3.3 All the prefetch suggestions coming from the L1D prefetcher are placed in the L1D Cache.

2nd Level Cache: For the L2C, we are using the enhanced SPP+PPF approach described in Section 3. Prefetches originating from the L2C can be placed in L2C or LLC, depending on the confidence estimate given by the perceptron sum.

Last Level Cache: The LLC prefetcher is the basic next-line prefetcher and does not incur any storage overhead. The prefetcher gets triggered on demand accesses and the prefetch accesses originating from the L1 prefetcher.

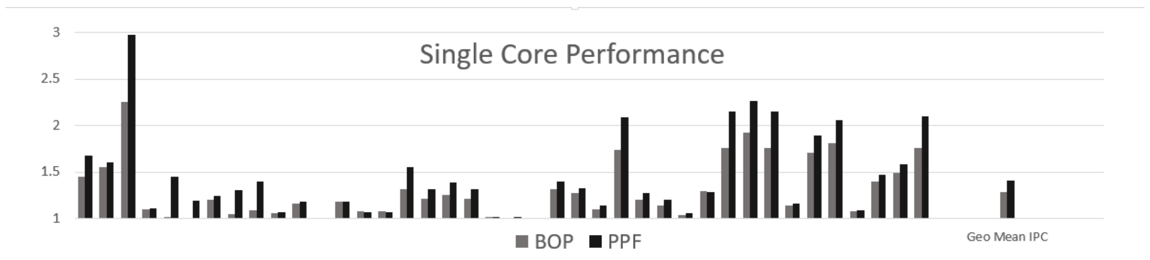

Figure 5: Single Core Performance

Structure	Entry	Components	Total
L1D Prefetcher			
Throttler	1024	Tag (7 bits) Score (5 bits)	12288 bits
Pages Accessed	8	Page number (52 bits)	416 bits
Overall L1D: 12,704 bits = 1.55 KBs			
L2C Prefetcher			
Signature Table	256	Valid (1 bit) Tag (16 bits) Last Offset (6 bits) Signature (12 bits) LRU (6 bits)	11008 bits
Pattern Table	2048	C_{sig} (4 bits) C_{delta} (4*4 bits) Delta (4*7 bits)	98304 bits
Perceptron Weights	4096*4 2048*2 1024*2 128*1	5 bits	113280 bits
Prefetch Table	1024	85 bits	87040 bits
Reject Table	1024	84 bits	86016 bits
Global History Register	8	Signature (12 bits) Confidence (8 bits) Last Offset (6 bits) Delta (7 bits)	264 bits
Accuracy Counters	1 1	C_{total} C_{useful}	10 bits 10 bits
Global PC Trackers	3	PC_1 (12 bits) PC_2 (12 bits) PC_3 (12 bits)	36 bits
Pages Accessed	8	Page number (52 bits)	416 bits
Overall L2C: 396,384 bits = 48.39 KBs			
Total: 409,088 bits = 49.94 KBs			

Table 1: Single-core Prefetcher Hardware

Single-Core Complexity: Table 1 shows a detailed analysis of the hardware overhead required to implement the three prefetchers. It is well within the championship budget of 64KB. In terms of L2C prefetcher's logical complexity, SPP is a cascade of three tables, with the output of one indexing into the next. Constructing the signature only requires simple operations like shifting and XOR. PPF requires parallel indexing into nine different tables and adding nine 5-bit integers, which is well within the complexity of currently implemented perceptron-based branch predictors. Weight updates are done only in steps of +1 or -1.

4.2 Multi-Core Configuration

When in a multi-core configuration, we disable the L1D Prefetcher, keep the same L2C Prefetcher and switch the LLC prefetcher altogether to SPP, without the enhanced PPF components. We do this by leveraging the information from NUM_CPUS parameter. We observed that the efficient filtering mechanism in PPF helps avoid pollution in the shared LLC and a further attempt to do aggressive prefetching leads to performance degradation. Hence we re-tune SPP's thresholds towards the safer side and this helps further boost the overall performance.

Multi-Core Complexity: Beside the single-core L2C prefetcher complexity, LLC SPP adds an overhead of 14.44 KBs per core, making a total of 62.83 KBs per core, which is within the budget allowed by the championship.

5. RESULTS

We test our implementation in the guidelines of the championship, using the memory intensive simpoints from SPEC 2017 traces. Figure 5 shows the single-core speedup obtained by our approach on the traces provided, along with the the geomean speedup. For reference, we've compared our results with the L2C only BOP [8], which was the winner of the 2nd DPC. Our prefetchers yield a speedup of 40.4% and 20.3% on single-core and multi-core systems respectively. This is 12% and 5.2% above BOP.

6. REFERENCES

[1] J. Kim, S. H. Pugsley, P. V. Gratz, A. L. N. Reddy, C. Wilkerson, and Z. Chishti, "Path confidence based lookahead prefetching," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, Oct 2016.

[2] D. A. Jiménez and C. Lin, "Dynamic branch prediction with perceptrons," in *Proceedings of the 7th International Symposium on High Performance Computer Architecture (HPCA-7)*, pp. 197–206, 2001.

[3] D. Tarjan and K. Skadron, "Merging path and gshare indexing in perceptron branch prediction," *ACM Trans. Archit. Code Optim.*, vol. 2, pp. 280–300, Sept. 2005.

[4] E. Teran, Z. Wang, and D. A. Jiménez, "Perceptron learning for reuse prediction," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, (Piscataway, NJ, USA), pp. 2:1–2:12, IEEE Press, 2016.

[5] D. A. Jiménez and E. Teran, "Multiperspective reuse prediction," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, (New York, NY, USA), pp. 436–448, ACM, 2017.

[6] E. Bhatia, G. Chacon, E. Teran, S. Pugsley, P. Gratz, and D. A. Jiménez, "Perceptron-based prefetch filtering," in *46th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 1-6, 2019 (to appear)*, 2019.

[7] A. J. Smith, "Sequential program prefetching in memory hierarchies," *Computer*, vol. 11, pp. 7–21, Dec 1978.

[8] P. Michaud, "Best-offset hardware prefetching," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 469–480, March 2016.