A METHOD FOR COMBINING PREFETCHING ALGORITHMS USING ORCHESTRATION TO IMPROVE OVERALL ACCURACY, COVERAGE AND ADAPTIVITY ON VARIABLE ADDRESS STREAMS

code available at: https://github.com/HarshVardhanKumar/HPCA_243/

HARSH KUMAR LEONIE GALLOIS

Abstract: Prefetcher performance has two important metrics: accuracy and coverage, which do not agree all the time. An overly aggressive prefetcher may improve coverage at the cost of reduced accuracy - harming the performance. Perceptron-based prefetch filtering is a way to increase the coverage generated by an underlying prefetcher without negatively impacting accuracy.

In this work, we extend the Perceptron-based filtering idea to more than one prefetcher. We propose a mechanism to arbiter these underlying prefetchers to capture complex variations in the memory access patterns. Using prefetch request from different prefetchers improves our ability to capture very complex program behaviour since two prefetchers using different criteria to predict prefetch addresses see the program behaviour in different ways. This improves our overall understanding of the address patterns. Arbitrating among the prefetch request of these prefetchers gives us a chance to cover most of the cache misses.

I. INTRODUCTION

The “Memory Wall” [1], the vast gulf between processor execution speed and memory latency, has led to the development of large and deep cache hierarchies over the last twenty years. Although processor frequency is no-longer on the exponential growth curve, the drive towards ever greater DRAM capacities and off-chip bandwidth constraints have kept

this gap from closing significantly. Cache hierarchies can improve average access times for data references with good temporal locality, however, they do not help to decrease the latency of the first reference to a particular memory address.

Prefetching is a well-studied technique which can provide an efficient means to improve the performance of modern microprocessors. The aim of prefetching is to proactively fetch useful cache lines from further down in the memory hierarchy, ahead of their first demand reference[3].
It is a very complex problem, however, that has several impacts on the first order performance of the processor. For example, where the prefetcher is located in the memory hierarchy will constrain the information available to it. Prefetching from DRAM into the lowest level of the cache, therefore, introduces a number of unique challenges.[3]

Some of these are typical: First, in most high volume CPU designs, the program counter (PC) is unavailable at lower level in the cache hierarchy. This can make PC-based patterns difficult to detect. Second, any prefetcher located at the last level cache must deal with physical addresses directly without the benefit of a TLB or other page table information. This means that address patterns must be discovered using only sequences of physical addresses. Since virtual to physical page mapping is often arbitrary, easily predictable sequences of virtual addresses spread over different pages may not exhibit discernible patterns after translation to the physical address space. This presents a particularly challenging problem for prefetchers that rely on discovering common deltas between consecutively requested addresses and applying them to future requests. [3]

In some earlier proposed prefetching techniques, the prefetching opportunity is limited to waiting until a cache miss occurs, and then prefetching either a set of lines sequentially following the current miss [2], a set of lines following a strided pattern with respect to the current miss [4], or a set of blocks spatially around the miss [5], [6]

In nearly all the prefetchers, there is always a trade-off between the two 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. Some of the previous approaches [8] try to balance this approach using perceptrons.

They basically use a perceptron filter on top of an underlying prefetcher to discard the seemingly not-so confident prefetches suggested by the underlying prefetches. They make the underlying prefetcher more aggressive to make many more prefetches. This will increase the coverage of the prefetcher. However, since the confidence mechanisms of the underlying prefetcher is reduced, it's possible that many of the suggested prefetches will not be accurate. Thus, accuracy will be reduced. To deal with this, they use perceptron layer to determine if the suggested prefetch is good enough to be used.

In this project, we aim at two things: Studying the performance of [8] and then try to extend their approach to multiple scenarios like
1. Using Perceptron filter on top of Spatial prefetchers like Bingo [9], delta prefetchers like Berti [10].
2. Implement the perceptron filter to combine and arbitrate prefetching from more than one underlying prefetcher and study its performance.

In this project, we explain how we can extend the approach taken by paper[8] to more than one prefetchers. We implement our model and test and compare its performance with other prefetchers on various benchmarks.

II. 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.

UNDERLYING PREFETCHERS:

1. SIGNATURE-PATH PREFETCHER

SPP [3] is the state-of-the-art delta-based prefetcher. It makes use of a signature comprising solely of up to four recent consecutive address deltas observed in a 4KB page. Each signature tracks at most four possible next deltas as prefetch candidates, along with a confidence value associated with each candidate.[11]

They use "Signature Table", "Pattern Table" and "Global History Register" for storing the signatures associated with the last 4 memory access pattern, then use the signature to index into Pattern Table to get the predicted delta patterns. The Global History register tries to learn from the prefetch suggestions that were rejected since they crossed the page boundary.

The confidence of each new prefetch candidate is a cascaded product of confidences leading to the candidate's level and the candidate's stored confidence value. A prefetch delta candidate whose cascaded confidence value is above a threshold value triggers a prefetch.[11].

2. ENHANCING SPP WITH PERCEPTRON PREFETCH FILTERING

In this approach the paper by E. Bhatia et al.[8] suggests to improve the aggressiveness of spp prefetcher and reject those prefetch suggestions which do not seem likely to be useful. They do some modifications in the spp like 1. Adjusting the aggressiveness, 2. Adding Prefetch and Reject filters, 3. Exporting data between the SPP and perceptron. They also define their state-of-art perceptron filter having nine different features for deciding upon whether to allow a prefetch or not.

3. BERTI PREFETCHER

This prefetcher was one of the finalists of DPC3 championship. In their paper[12], Alberto Ros suggest some modifications upon BOP[13] prefetcher to improve timeliness of the prefetches.

The Berti Prefetcher finds the delta that provides the best timeliness for memory blocks in each page. The prefetcher works in two modes: (i) on the first access to a block, in a certain period of time, the prefetcher issues a request for the next block according to the best delta found; (ii) for cold pages, a burst mechanism fetches blocks that cannot be reached adding the delta to the current accessed block.[10]

Its mechanism is highly inspired by BOP[12] is based on the observation that timeliness and best delta varies from page to page, and a global delta results in missing opportunities. They try to find a unique delta per page that provides the best timeliness for its blocks and applies it to the next accesses[10].

4. PERCEPTRON LEARNING

Perceptron learning for microarchitectural prediction was originally introduced for branch prediction [13][8]. The hashed perceptron predictor hashes several different features into values that index into several feature tables. Small integer weights are learned by the perceptron for each of the features per prefetch request.

These weights are then read and summed from the tables. If the sum exceeds a certain threshold, a positive prediction is made, otherwise, a negative prediction is made. Once the ground truth is known, the weights corresponding to the prediction are updated based on whether the prediction was correct or not.

III. PREFETCHING ENHANCEMENTS

1. CHANGES MADE TO BERTI PREFETCHER

Increasing the prefetching width is not always beneficial as it may result in problems of cache pollution. However, if we have some mechanism to ensure that only useful prefetches make their way to cache regardless of prefetch suggestion, then allowing a large prefetch depth may be beneficial.

Also, some useful prefetches are generated long after the confidence of the underlying prefetcher has fallen below the point at which prefetcher inaccuracy would lead to performance degradation - that is, coverage may continue to increase far beyond the point at which accuracy stops.[8] Thus, in order to allow deep speculation in the prefetcher, inaccurate prefetches must be filtered out.

Eshan Bhatia et al.[8] propose perceptron-based learning as a mechanism to differentiate between potentially useful deeply speculated prefetches and likely not-useful ones. In their approach, they place a Perceptron based Prefetch Filter(PPF) between the prefetcher and the prefetch insertion queue to prevent not-useful prefetches from polluting the higher levels of memory hierarchy.

In our work, we've used the same approach, but with the Berti prefetcher. First, we modify the baseline Berti design so as to de-throttle it, enabling higher coverage. We tuned down the internal throttling mechanism to allow all the prefetches to pass through. In particular, the following changes were made:

- We removed the throttle on no. of burst requests, thus increasing no. of burst prefetches.
- We removed the throttling based on current_pages_table. The prefetch request are not limited by the contents of current_pages_table.
- We removed the dependency on Berti value confidence.
- We introduce lookahead mechanism in the Berti Prefetcher.

2. THE PERCEPTRON FILTER

We leverage the same microarchitecture of PPF, as well as the steps required to filter out not-useful prefetches from E.Bhatia et. al[8]. The perceptron filter is organized as a set of tables, where each entry in the table holds a weight. Each table is indexed using values from the corresponding feature.

The prefetcher (Berti) is triggered on every demand access to L2 cache. The suggested prefetch is then analysed using the perceptron filter to decide its usefulness and fill-level as in [8].

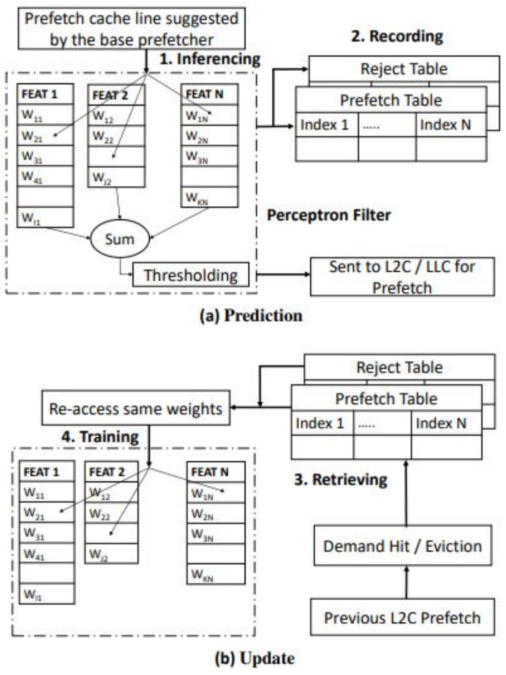

Fig: Perceptron Data Path and Operation[8]

Perceptron training is triggered whenever a prefetched block leads to a demand hit or a cache eviction. A uniform update rule is followed - for a correct prediction, the weights are incremented by one. In the case of cache evict without used, the weights are updated in opposite direction.

IV. PROPOSAL

Inspired by the Bouquet prefetching paper[14], we decided to improve the perceptron-based filter prefetching by combining several kinds of prefetchers. To get the best performance out of them, we switch between them based on the previous pattern of prefetches. One approach is to allow prefetch of all prefetch-suggestions by the underlying prefetchers and update the weights of all the prefetcher features in their respective perceptron filter tables. In that way, all the prefetch methods are running concurrently and are using every chance to adjust themselves correctly.
The drawback can be that if the two prefetch methods are having close accuracy value, then the two suggestions could interfere with each other when placed in the cache. One prefetch could evict another while entering the cache, for example.

To decide when to switch to the other prefetcher, another approach is to compare the resulting weighted sum of each feature for all prefetch suggestions. The one with higher value is prefetched while the other one is abandoned. Now, only one prefetch is conducted at a time so no interfering can happen.

However, the weights of one prefetcher are updated only when it is chosen. The chances these weights get to be adjusted are now decreased. The worst case scenario could be that the weights of one prefetcher are really far from their optimized value to get exact prefetches and the weights of the second prefetcher close to their ideal values.

To avoid this unpleasant scenario, we propose a third approach. The third method consists also of choosing only one prefetch suggestion as the previous approach does. This time, however, we switch to the second prefetcher if in the last request, the prefetch was unused by the cahe. In this case, we expect to switch many more times between the two prefetchers than the previous method. The weights of both perceptron filter tables will be updated each time the other prefetcher failed to produce a correct prefetch. This last method actually embraces the problem by choosing the least bad prefetcher rather than choosing the best one.

For this project, we selected the second approach, as this is easy to implement and lightweight for two prefetchers.

In our work, we implement the perceptron filter over two underlying prefetchers: SPP and Berti - one is good for lookahead prefetching while the other one is good for timely prefetches.

1. IMPLEMENTATION

This section describes our implementation of PPF and the range of features used to determine the usefulness of prefetches. We have selected two prefetchers SPP and Berti for underlying prefetchers.

More specifically, we implement two perceptrons - one for each of the prefetchers and then use the prediction returned by both the perceptrons to arbiter between the two prefetchers. The perceptrons corresponding to each prefetcher uses the metadata specific to that prefetcher. For example, the perceptron for SPP uses lookahead depth, signature and confidence counter, whereas the perceptron related to Berti uses the First_offset, Berti_value and Berti_confidence.

FEATURES USED BY PERCEPTRON

For the SPP part, we leverage the features listed by E.Bhatia et. al[8] for the perceptron. However, for the Berti part, defining the features was tricky. We needed to select those features which were independent of the specific prefetch request as well as which were good enough to find out correlation among prefetches.

We use the following set of features for Berti prefetcher:

- **Physical Address:** The lower bits of address of demand access. This value corresponds to a stream of accesses that have been seen before. Thus, the PPF will correlate the past behaviour of this address to the prefetch outcome.[8]
- **Cache line , Page Address**
- **Berti_confidence XOR Page Address:** By itself, Page Address is not a good basis for filtering. This feature mixes the confidence of Berti_value for that page.
- **IP & First_offset:** It measures the correlation of IP of cache miss with the First_offset of the physical address.
- **ip1 XOR (ip2>>1) XOR (ip3>>2):** Here, ipi refers to the last ith IP before the instruction that triggered the current prefetch. Hashing the last three IPs inform the perceptron about the path that led to the current demand access.
- **Berti_confidence XOR IP XOR current_berti:** This correlates the Current Berti value alongwith its confidence to the IP.
- **N_Delta:** We implement in a Global History Register, the last 32 ip values and last 32 addresses. For the current prefetch suggestion, we calculated the delta1 = pf_addr - last_address and compare delta1 with the deltas of pairs of last addresses. No. of matches found is stored. Please note that this delta is calculated on the physical addresses and thus, matching no. of deltas helps to find out how strongly the current prefetch is related to the current page.
- **Berti_confidence:** The berti_confidence is stored for each berti_value. Note that berti_value is common to all the acceses to a page. While the original Berti_confidence does not directly make the decision to prefetch, the perceptron correlates it to the correctness of prefetch to any given page using the Berti_confidence.

METHODOLOGY

PERFORMANCE MODEL

We use the ChampSim simulator for the evaluation of our approach against prior approaches. ChampSim is an enhanced framework that was used for the 3rd Data Prefetch Championship (DPC3). We model 1-core out-of-order machine. The details of the configuration parameters are summarized below.

Block	Configuration
CPU Core	1-core, 4GHz 256 entry ROB, 4-wide
Private L1 D cache	32 KB, 8-way, 4cycles, 8MSHRs, LRU
Private L2 Cache	256 KB, 8way, 8 cycles, 16 MSHRs, LRU, Non-inclusive
Shared LLC	2 MB/core, 16 way, 12 cycles 32 MSHRs, LRU, Non-inclusive
DRAM	4GB 1-channel (single-core) 8GB 2-channels (multi-core) 64-bit channel, 1600MT/s

The block size is fixed at 64 bytes. Prefetching is only triggered upon L2 cache demand accesses but could be directed to the L2 or last-level cache. No L1 data level prefetching is done. The LRU replacement policy is used on all levels of cache hierarchies. Branch prediction is done using the perceptron branch predictor. The page size is 4KB. ChampSim operates all the prefetchers strictly in the physical address space.

WORKLOADS

We use five workloads available in the SPEC CPU 2017 suite.

PREFETCHERS SIMULATED

We simulated the original [8] code, original berti code, Bingo[9]. We then modified Bingo and Berti code to make them more aggressive and again simulated them.

After this, we implemented Perceptron on top of Bingo and Berti separately, then combined Berti and SPP with two separate perceptrons and measured their performance. The evaluation results are attached as a separate file.

EVALUATION and ANALYSIS

To evaluate our proposal, we use prefetch accuracy that is in our case the percentage of the useful prefetches over the requested=issued ones. We also look upon the miss latency in number of cycles. The miss latency is on average around a hundred cycles. On the other hand, the prefetch accuracy varies a lot, from zero to almost forty.

We do not have any specification or description of the benchmarks. So we pick them randomly.

With three different benchmarks: 605mcf_s1554B, 607cactuBSSN_s4004B and 607cactuBSSN_s4248B we evaluated the prefetch accuracy and miss latency for three different implementations that are a next line prefetcher for l1d, same for l1i and llc and either berti or aggressive berti or aggressive berti with a perceptron filter for l2 prefetcher. (Warmup Instructions: 1000000
Simulation Instructions: 150000000
) Please refer to Appendix 1 and 2.

Here, in terms of prefetch accuracy, in general, berti aggressive performs better. In the second benchmark, it does not, berti and berti aggressive plus perceptron outperformed it. In the third and second benchmarks, berti and aggressive berti plus perceptron are performing similarly.
Concerning miss latency, for the third benchmark, aggressive berti is performing

the worst and the other two are similar. For the second benchmark, for l1, aggressive berti is the least efficient. Otherwise, it performs better. For the first benchmark, none real outperformer can be decided. So in general, concerning l1, aggressive berti is not the one to choose.

With three different benchmarks, 410.bwaves-945B, 602.gcc_s-734B and 605.mcf_s-1554B and a number of warmup instructions of 1000000 and a number of simulation instructions of 50000000, we evaluated the approach with l1d next line, l1i next line, l2 with berti plus perceptron filter and spp plus perceptron filter -switching between the two-, llc next line. Please refer to Appendix 4.

In this evaluation, we have a few times, no prefetch requested. That is why, several values appeared to be null. The values obtained here are very different. We can only point out that only three benchmarks is a too small number to extrapolate a conclusion from them. As a few benchmark as zero prefetch requests, their miss latency is also null.

With one benchmark, 410.bwaves-945B, 1000000 Warmup Instructions and 150000000 Simulation Instructions, we evaluated the prefetch accuracy and miss latency of this approach: l1i next line, l1d next line, either berti, spp with perceptron filter or bingo and llc next line. Please refer to Appendix 3.

Except in llc, berti is performing better here. In llc, spp and bingo are similar. Otherwise, bingo performed lower than spp.
For miss latency, bingo performed badly compared to the other two in l1. Everywhere else, the results are quite similar for the three approaches.

We also implemented a prefetcher with next line as l1i, bingo as l1d, X as l2 and next line as llc. (X is either one Berti approach or bingo or spp plus perceptron)

CONCLUSION

We extended the perceptron filter as in [8] over two different prefetchers - Berti and Spp and measure the performances. We found out that the perceptron based filter prefetch is very sensitive to the underlying prefetcher. Also, our method is only implemented in L2 (we considered LLC prefetcher as next line). In our work, L1 has a very naive prefetcher that consequently reduces the overall efficiency. We also tested bingo [9] prefetcher in L1 and the results improved substantially. We need to test our method on numerous other benchmarks.

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]: A. J. Smith, “Sequential program prefetching in memory hierarchies,” Computer, vol. 11, pp. 7–21, December 1978.

[3]: M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, “Efficiently prefetching complex address patterns,” in Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture, 2015.

[4]: T. Chen and J. Baer, “Effective hardware-based data prefetching for high-performance processors,” IEEE Transactions on Computers, vol. 44, pp. 609–623, 1995.

[5]: S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, “Spatial memory streaming,” in ACM SIGARCH Computer Architecture News, vol. 34, pp. 252–263, IEEE Computer Society, 2006

[6]:I. Hur and C. Lin, “Memory prefetching using adaptive stream detection,” in Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 397–408, IEEE Computer Society, 2006

[7]:https://github.com/bakhshalipour/Bingo

[8]:Eshan Bhatia, Gino Chacon, Seth Pugsley, Elvira Teran, Paul V. Gratz, and Daniel A. Jiménez. 2019. Perceptron-based prefetch filtering. In Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19). ACM, New York, NY, USA, 1-13. DOI: https://doi.org/10.1145/3307650.3322207

[9]:M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran and H. Sarbazi-Azad, "Bingo Spatial Data Prefetcher," *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Washington, DC, USA, 2019, pp. 399-411.
doi: 10.1109/HPCA.2019.00053

[10]: Alberto Ros, "Berti: A Per-Page Best-Request-Time Delta Prefetcher". The 3rd Data Prefetching Championship, Phoenix, AZ (USA), June 2019.

[11]: Rahul Bera, Anant V. Nori, Onur Mutlu, and Sreenivas Subramoney. 2019. DSPatch: Dual Spatial Pattern Prefetcher. In Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52). ACM, New York, NY, USA, 531-544. DOI: https://doi.org/10.1145/3352460.3358325

[12]: Pierre Michaud. A Best-Offset Prefetcher. 2nd Data Prefetching Championship, Jun 2015, Portland, United States. hal-01165600

[13]: 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.