MPMLP: A Case for Multi-Page Multi-Layer Perceptron Prefetcher

Ali Asgari* Andrew Gunter* Mehdi Saeidi* Mieszko Lis Prashant Nair
The University of British Columbia
{aasgarik,agunter,msaeidi,mieszko,prashantnair}@ece.ubc.ca

***Abstract*—As computing systems scale their memory systems continue to be limited by latency and bandwidth. Data prefetching is an efficient technique that can be used to efficiently utilize the memory bandwidth while mitigating memory latency. While modern data prefetchers can efficiently predict regular access patterns, they are inefficient in predicting complex access patterns that span across page boundaries. This paper proposes an ML-based LLC data prefetcher called Multi-Page Multi-Layer Perceptron prefetcher (MPMLP). The MPMLP prefetcher contains two sub-prefetchers; namely a traditional Best-Offset Prefetcher and an MLP-based prefetcher. The MPMLP prefetcher is geared to predict regular and complex patterns across page boundaries. For benchmarks that the Best-Offset Prefetcher does not perform well, the MPMLP prefetcher tries to learn the data access pattern at LLC using a Multi Layer Perceptron (MLP). Overall, the MPMLP prefetcher provides a speedup of 32% as compared to a baseline that does not employ prefetching.**

I. Introduction

The advent of multicores and accelerators have been instrumental in scaling compute throughput in modern Von-Neumann machines. However, their memory latency and bandwidth have not scaled proportionately. To overcome this hurdle, data prefetching is seen as an efficient technique to utilize the available memory bandwidth and reduce or eliminate memory access latencies. The goal of the prefetcher is to accurately bring-in useful data blocks in a timely manner when the memory bandwidth is unused [14]. However, as workloads evolve, their sequence of memory accesses have become complex and prefetching useful data blocks has become increasingly difficult [3]. To overcome this concern, this paper proposes a machine-learning (ML) based hardware data prefetcher for the Last Level Cache (LLC).

This paper observes that traditional prefetchers, such as the Best-Offset prefetcher, are efficient in prefetching data blocks that are a part of a dominant access pattern. However, modern workloads also exhibit hard-to-predict (complex) access patterns that could span across multiple physical pages. Furthermore, as modern operating systems can randomize the physical page locations for contiguous virtual pages, prefetching these data blocks is challenging. Therefore, this paper uses a hybrid approach to data prefetching.

Our hybrid data prefetcher, called as the Multi-Page Multi-Layer Perceptron Prefetcher (MPMLP), is made up of two sub-prefetchers; namely a multi-layer perceptron-based sub-prefetcher and the Best-Offset (BO) sub-prefetcher [4, 7]. The sub-prefetchers are designed to share the prefetching budget. Furthermore, the MPMLP prefetcher is equipped to issue prefetch requests to data blocks that are outside physical page boundaries. To enable this, MPMLP maintains a page transition table that keeps a record of the most recent page transitions for predicting the address of the next physical page. Our analysis shows that, on average, the MPMLP prefetcher outperforms the no-prefetcher baseline by 32% across SPEC2006, SPEC2017, and GAP benchmarks.

A. Motivation

Traditional prefetchers are typically designed to prefetch commonly occurring access patterns. For example, state-of-the-art prefetching techniques like the VLDP [12], Sandbox [10], Bouquet of Instruction Pointers [8], and the Best Offset (BO) prefetchers [7] predict offsets from the current access. The BO prefetcher exploits the idea that a given program phase will have some dominant (i.e., most commonly occurring) offset(s) between consecutive memory addresses.

However, when the workload lacks a dominant offset, an offset-dependent prefetcher like the BO prefetcher does not improve performance. For example, in the Single-Source Shortest Path (SSSP) algorithm, whenever a node *S* is processed, the metadata of all of its neighbours are processed one after another, and the neighbours of each node *S* are traversed several times as the algorithm is iterative. Since each node is connected to some other nodes with irregular indices, its neighbours are not stored with a dominant offset in the memory. The sequence of accesses to process a node *S* in SSS algorithm is shown in Fig. 1, where the x-axis refers to the offset at which the nodes are stored. As shown in the picture, each of the accesses happens in an *unintuitive* offset that cannot be captured by a BO prefetcher.

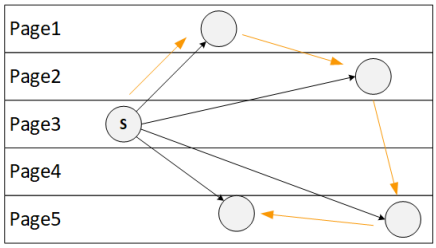

Fig. 1. Memory accesses to process the node *S* of a graph according to the Single-Source Shortest Path (SSSP) algorithm.

Prefetchers can also be designed to not rely on detecting and replicating offsets. For example, Markov prefetchers and

*These authors contributed equally – ordered in the ascending order according to the first letters of their last names.

Dependence Prefetchers [6, 11] can be used to capture the complex node traversal patterns in the SSSP application. However, these prefetchers would require a large amount of storage to store the state information such as path confidence and produce-consumer pairs respectively.

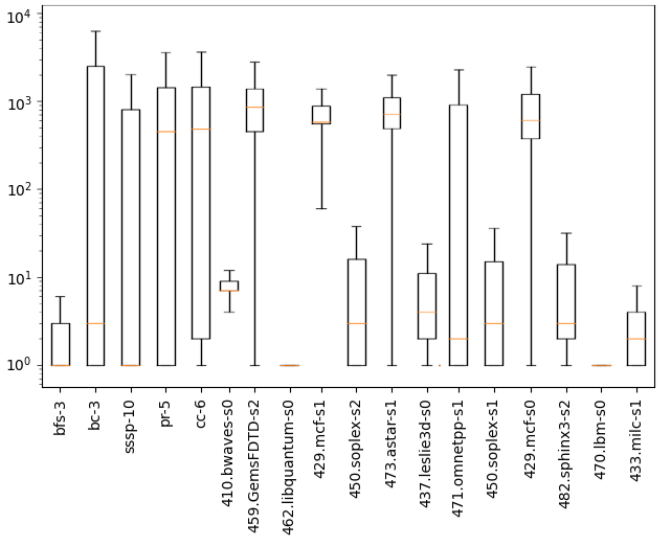

Fig. 2. Number of accesses it takes to get to the same page. This value for many of the benchmarks is >1; implying that there are several transitions to other pages before referencing a data block in the same page. This helps motivate the need to prefetch across pages.

Furthermore, complex data access patterns can span across pages. For instance in the benchmark SSSP, as shown Fig. 1, when accessing the node *S* itself and its neighbours' metadata, four different pages has been accessed. As shown in Fig. 2, in several benchmarks transitions between pages happen quite frequently as the number of memory accesses into different pages while accessing subsequent lines in a page is >1.

Prefetching data blocks is further complicated by the fact that, after virtual to physical address translation, the operating system (OS) may assign contiguous virtual pages into seemingly random physical pages. Therefore, one of the major challenges is handling prefetch requests that span across physical page boundaries. As the next data address can be from any seemingly random physical page, it is difficult for a traditional prefetcher to predict what makes the upper bits of the next address (the physical page number).

In contrast to the conventional prefetchers, a well-trained ML-based prefetcher does not need as much guidance or storage overheads for identifying pattern(s) that can be exploited within the target program. This is because machine learning models are capable of finding statistical trends that aid the minimization of a loss function supplied to the model. An example of these statistical trends is the repetitive traversal of the same neighbours (see Fig. 1). Importantly, neural networks can learn non-linear relationships between their inputs and the loss function and deep networks will form a hierarchical representation of knowledge between layers of the network. As a result, unlike traditional prefetchers, a neural network prefetcher is capable of memorizing numerous frequent data access patterns in a space efficient manner and does not need to maintain a large amount of information state [6]. For this reason, the proposed MPMLP prefetcher opts to use a multi-layer perceptron as its ML-based prefetcher.

However, data access patterns that tend to be recurring, the virtual to physical page mappings can be completely random. Due to this random nature of page mapping, the physical page address that is addressed after the current page cannot be predicted using an ML based approach. Therefore, to enable prefetching across page boundaries, the MPMLP prefetcher must be designed with a tracking table to help predict subsequent physical page addresses.

II. Prefetcher Design

As we designed the prefetcher for the ML-Based Data Prefetching Competition [2], our design is constrained by the requirements of this competition. We focused on the following shortcomings of existing prefetchers:

- They are unable to exploit complex frequent patterns. To address these patterns we use MLP.
- The next address to prefetch may happen in the next virtual page, with an unknown physical address. To solve this we propose use of a page transition table (PTT).
- Many applications have simple behaviours with dominant address offsets which our MLP model is not capable of fully and quickly capturing those offsets. To exploit these patterns as well, we complement our prefetcher with a Best Offset prefetcher [7].

As the competition environment does not provide the data prefetcher any access to the OS page management mechanisms, we also need a hardware-only mechanism to predict physical pages that exhibit temporal locality. We treat this prediction task as orthogonal to the underlying task of data prefetching, i.e., separate the task of predicting the cache set index from the task of predicting the next physical page to be accessed. To predict page numbers, we supplement the MPMLP prefetcher with a page transition table that records the last page transition at a page boundary for each Instruction Pointer (IP), and use this information for predicting the next page when a page transition is predicted.

Applications tend to have varied memory access behaviours. For instance, some applications tend to have highly irregular memory accesses while other applications may follow a relatively regular access pattern. In case of irregular memory accesses, a prefetcher such as the BO prefetcher that relies on the continuation of phase-based patterns may perform poorly. However, in such a scenario, a multi-layer perceptron (MLP) based prefetcher may be better suited to finding *unintuitive* patterns despite access irregularity. In case of regular memory accesses, a phase-based BO prefetcher may be highly effective while a MLP-based prefetcher is unnecessarily difficult to train. This leads us to attempt using our MPMLP model in tandem with an auxilary BO model.

A. Overview

Fig. 3 shows the components of the proposed prefetcher. The three major components are: an MLP-based prefetcher ②, an auxilary BO prefetcher ③ [7], and a page transition table ④. Each time a memory access occurs, the MLP prefetcher ② and the BO prefetcher ③ issue one prefetch request each. The MLP perfetcher receives its input from the Memory Access History ① unit, which keeps track of a window of a limited number of previous accesses of each IP. It then predicts an offset *within* a page, and whether the access would be in the current page or another page. If it predicts that another page is going to be accessed, the address of the next page is predicted according to the page transition table ④. Finally, the physical address is obtained by concatenating the page address, which is either equal to the current page or the page read from PTT, and offset within the page. The next two subsections describe the details of the MLP model and the page transition table.

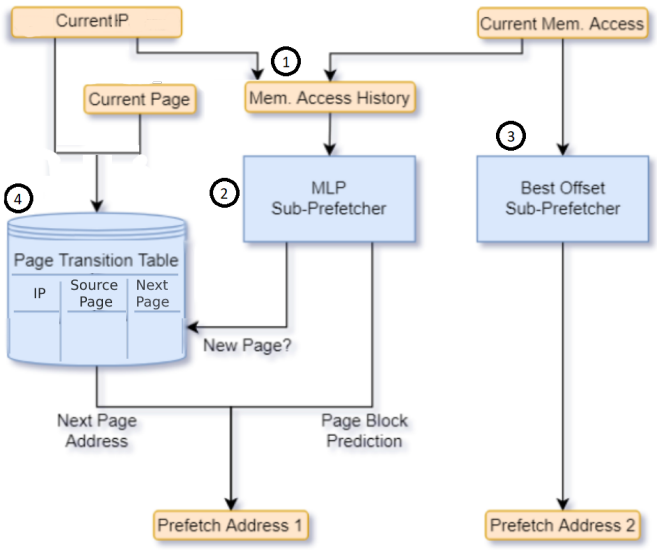

Fig. 3. The overview of the MPMLP prefetcher. The MPMLP prefetcher consists of 3 primary components. First, an MLP-based Prefetcher. Second, the Best-Offset Prefetcher. Third, a Page Transistion Table for tracking physical pages with temporal locality.

B. Multi-Layer Perceptron Prefetcher

We formulate the prediction task as a multi-class classification problem [5]. The prefetching problem is formulated as a prediction of the index of cache lines within a page to be accessed in the future by a given instruction pointer (IP). This is based on the insight that, rather than as a direct data address prediction based on the raw LLC access stream, the IP helps differentiate between parts of code that may have different prefetch characteristics for the same data address. The structure of the MLP prefetcher is shown in Fig. 4.

When a memory access is encountered, the set indices of the last h memory accesses for this IP (including the current one) are encoded as one-hot vectors, and the vectors are OR'd together (i.e., an embedding bag). For a cache with s sets,

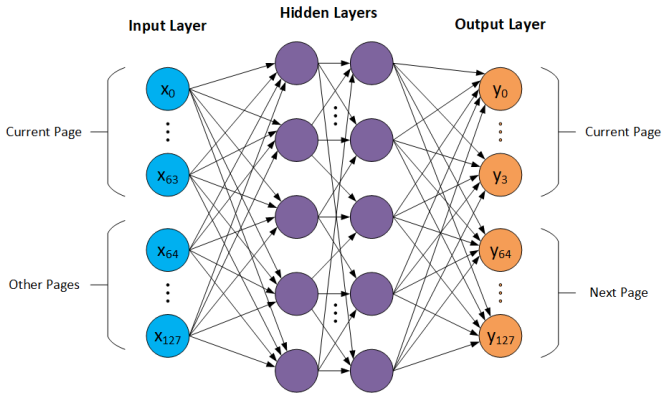

Fig. 4. The structure of the MLP sub-prefetcher. The MLP sub-prefetcher consists of input layer, 2 hidden layers, and output layer.

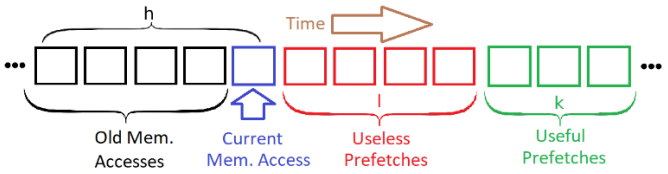

Fig. 5. The window of accesses considered for training. On the left you see the h accesses used as history, i.e. input to the model. On the middle you see the l skipped accesses to reach some timely target accesses. On the right the k target accesses are shown.

each vector has $2s$ elements: the first s elements correspond to accesses on the same page, and the second s elements to accesses on any other page. (In the configuration we evaluate, $h = 4$ and $s = 64$.)

Unlike [5], which tries to include the most frequently parts of the address space into the output space, the output space of our prefetcher only contains two pages, which are the current page and the next page. Thus, the output of our model is also a 128-element, as shown in Fig. 4. Each element is between 0.0 and 1.0, representing the likelihood that the corresponding cache line will be accessed in the future. The d most likelihood cache lines are selected to be prefetches.

Our training labels are also binary vectors of 128 elements. To obtain the output label, a window of k future are represented as binary vectors of 128 elements with the same method as representing the window of past accesses to obtain the input to the network. Then, these k vectors are OR'd together, similar to the way the input vector is obtained. Optimization uses a binary cross entropy loss function. In the competition framework, the training is done offline on some N instructions (100 million in our evaluations) instructions, and evaluation is done on the next N instructions of a given application trace.

One of the major concerns for prefetching is the timeliness of the prefetch requests. If we prefetch a cacheline too late (i.e., too close to the point in time at which the corresponding data is needed), then the prefetch request will not be useful as the demand access to this address will occur before the prefetch completes. This creates a need to predict far into the future, rather than predicting the next few accesses. We choose

to skip l accesses (see Fig. 5) into the future per instruction pointer and only look at the accesses that occur later than this value for training our model. We use $l = 5$ in the presented evaluations.

Note that this model actually disregards sequencing information: that is to say, the model does not know the order in which the cache lines were accessed in the last h accesses. While this throws away some information and reduces the potential peak performance of the model, for the investigated MLP variations, we find that it also allows for effective training and the model is still able to learn meaningful patterns in the LLC accesses of many IPs. This formulation of the input is highly amenable to the MLP's ability to derive important information from subtle statistical patterns.

C. Page Transition Table

In several applications, the same data structures tend to be accessed repeatedly, and thus, the sequence of accessed physical pages (albeit spread throughout the memory system) are also likely to repeat. Thus, for any load instruction, the most recent page to which a transition from the current page has occurred is used as the prediction for the next page accessed. This fact is also reflected in Fig. 2, which shows that after transitioning from a certain page to another one, in many cases that physical page is accessed again later. This observation is used by the MPMLP prefetcher by keeping track of these page transitions.

To keep track of the most recent page transitions, a page transition table is maintained for each of the IPs. Every time an IP accesses a physical page y right after accessing physical page x, the entry (x, y) is written to the page transition table corresponding to that IP. If there is already an entry with source x, it is over-written, and otherwise, a new entry is allocated in the table.

III. Methodology

We use PyTorch [9] to implement the MLP model. The model has 3 fully connected linear layers. We apply Dropout with rate 0.5 [13] for the last layer. The output of our MLP model has 128 neurons. The first 64 neurons represent data blocks in the current page whereas the other 64 neurons represent data blocks from the next page. The input layer also has the similar 128 neurons, with 64 neurons representing the current page and the other 64 neurons representing a page that was accessed before the current page (the h last accesses can come from multiple physical pages). The last layer has a *softmax* activation and the other layers come with ReLU activation. The intermediate layers of neurons have 376 and 400 neurons respectively.

As illustrated in Fig. 5 we have three windowing parameters, h, l, and k. We empirically found the values $h = 4$, $l = 5$ and $k = 2$ work best overall, and used them for all benchmarks. Most importantly, a relatively short history makes the learning process faster for the MLP prefetcher.

We use the the ChampSim [1] simulator to evaluate the MPMLP prefetcher. The efficacy of the MPMLP prefetcher is measured on SPEC2006, SPEC2017, and GAP benchmark traces that are provided by the Machine Learning Data Prefetching Competition [2]. In this paper, we compare the MPMLP prefetcher to a baseline system that does not employ prefetching. In this work, we train the MLP model on the first 100 million instructions of each benchmark and evaluate the IPC for the next 100 million instructions.

IV. Results

Fig. 6 shows the performance of three prefetchers for 40 random traces out of the 99 benchmarks provided by the data prefetching competition. On average, the MLP-only prefetcher can provides a speedup of 25% (geometric mean) over a no-prefetcher baseline. The Hybrid MPMLP prefetcher can increase this speedup to 32% taking advantage of both BO prefetcher [7] and the MLP prefetcher.

The most benefits for the MPMLP prefetcher are for the *607.CactusBSSN* benchmark. Both the MLP and MPMLP models also outperform BO in *sssp* benchmarks. As in the example from Section I, that is because the there is no dominant offset in the memory accesses of this benchmark, but the sequence of accesses are repeated several times, and thus, MPMLP can recall the memory access patterns from the training instructions. In addition, its page transitions are repeated several times. Therefore, the PTT can predict the next page that is going to be accessed, while the BO prefetcher only prefetches within the same page.

In the *654.roms* benchmark, the MPMLP model prefetcher performs better than each of the BO and MLP prefetchers. That is because in this benchmark, some IPs have regular access while the others have regular access patterns. As a result, a mixture of BO and MLP captures both regular and irregular accesses, which results in a higher performance than prefetchers that focus on only one those patterns.

When offsets are regular and predictable, however, the MLP component offers no advantage over BO: for example, in *649.fotonik3d*, BO-only performs better than MLP-only.

V. Conclusions

As memory bandwidth and latency becomes increasingly important, data prefetching has become an important area of research. To this end, this paper showcases the MPMLP prefetcher that is hybrid prefetcher consisting of a Best Offset Prefetcher and an MLP-based prefetcher. The Best-Offset prefetcher is useful in prefetching regular accesses. On the other hand, the MLP-based prefetcher is found to be useful to prefetch irregular accessed. To improve performance further, the MPMLP prefetcher is designed to prefetch across page boundaries. This helps improve the timeliness of the MPMLP prefetcher while also enabling a larger coverage. Overall, across SPEC2006, SPEC2017, and GAP benchmarks, the MPMLP provides a speedup of 32% (on average) as compared to a no-prefetcher baseline.

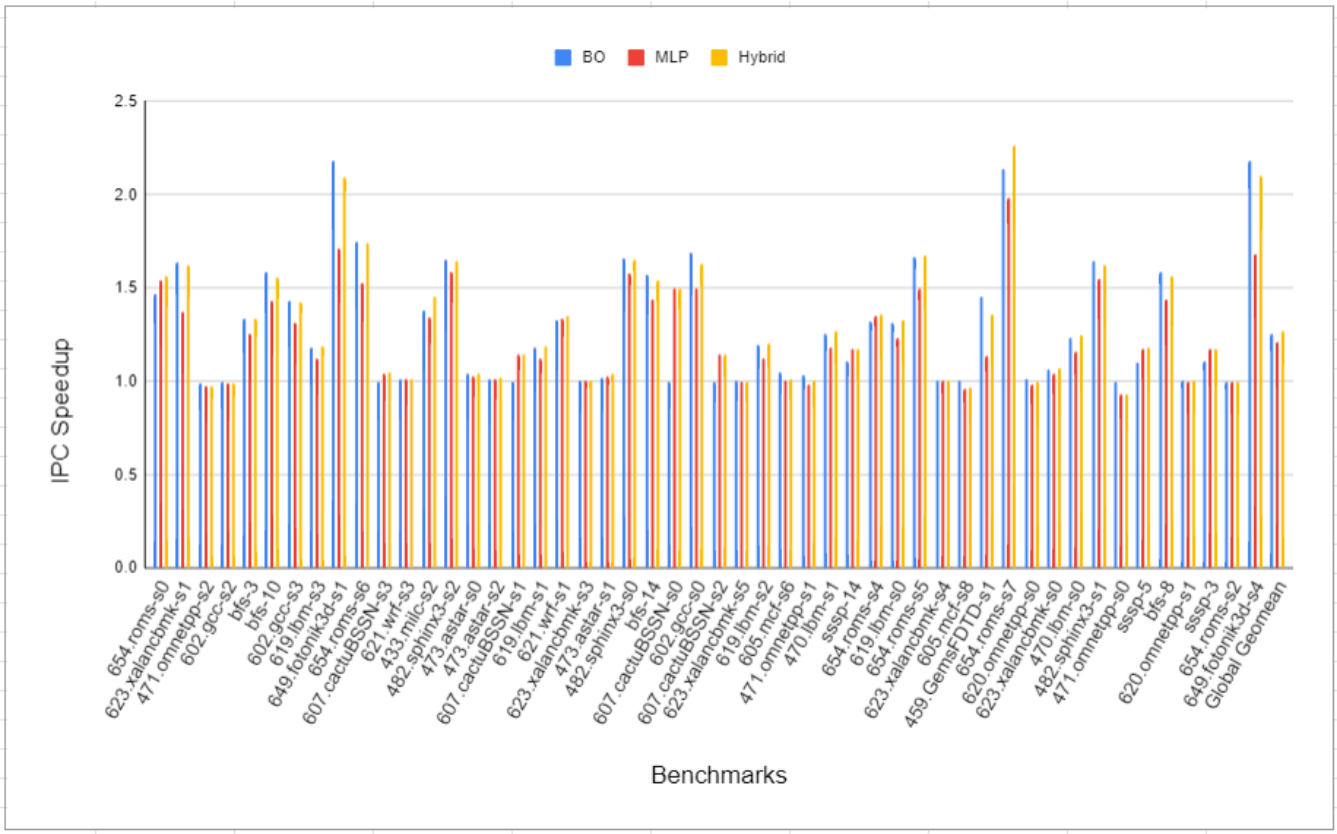

Fig. 6. Speedup over a no-prefetcher baseline for MLP prefetcher, Best Offset prefetcher and the Hybrid (MPMLP) prefetcher of the two. Overall, MPMLP provides a speedup of 32% as compared to an MLP-only implementation that provides 25% speedup.

References

[1] "ChampSim," https://github.com/Quangmire/ChampSim, 2021, [Online; accessed 16-May-2021].

[2] "Machine Learning Data Prefetching Competition," https://sites.google.com/view/mlarchsys/isca-2021/ml-prefetching-competition?authuser=0, 2021, [Online; accessed 16-May-2021].

[3] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.

[4] R. Collobert and S. Bengio, "Links between perceptrons, mlps and svms," in *Proceedings of the Twenty-First International Conference on Machine Learning*, ser. ICML '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 23. [Online]. Available: https://doi.org/10.1145/1015330.1015415

[5] M. Hashemi, K. Swersky, J. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan, "Learning memory access patterns," in *Proceedings of the 35th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, J. Dy and A. Krause, Eds., vol. 80. PMLR, 10–15 Jul 2018, pp. 1919–1928. [Online]. Available: http://proceedings.mlr.press/v80/hashemi18a.html

[6] D. Joseph and D. Grunwald, "Prefetching using markov predictors," *IEEE Transactions on Computers*, vol. 48, no. 2, pp. 121–133, 1999.

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

[8] S. Pakalapati and B. Panda, "Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching," in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ser. ISCA '20. IEEE Press, 2020, p. 118–131. [Online]. Available: https://doi.org/10.1109/ISCA45697.2020.00021

[9] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[10] S. H. Pugsley, Z. Chishti, C. Wilkerson, P.-f. Chuang, R. L. Scott, A. Jaleel, S.-L. Lu, K. Chow, and R. Balasubramonian, "Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2014, pp. 626–637.

[11] A. Roth, A. Moshovos, and G. S. Sohi, "Dependence based prefetching for linked data structures," in *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS VIII. New York, NY, USA: Association for Computing Machinery, 1998, p. 115–126. [Online]. Available: https://doi.org/10.1145/291069.291034

[12] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, "Efficiently prefetching complex address patterns," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: Association for Computing Machinery, 2015, p. 141–152. [Online]. Available: https://doi.org/10.1145/2830772.2830793

[13] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, no. 1, p. 1929–1958, Jan. 2014.

[14] H. Wu, K. Nathella, D. Sunwoo, A. Jain, and C. Lin, "Efficient metadata management for irregular data prefetching," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 449–461. [Online]. Available: https://doi.org/10.1145/3307650.3322225