PATHFINDER: Practical Real-Time Learning for Data Prefetching

Lin Jia
University of Utah
Salt Lake City, Utah
lin.jia@utah.edu

James Patrick McMahon
University of Utah
Salt Lake City, Utah
u0975161@umail.utah.edu

Sumanth Gudaparthi
University of Utah
Salt Lake City, Utah
sumanth.gudaparthi@gmail.com

Shreyas Singh
University of Utah
Salt Lake City, Utah
shreyas.singh@utah.edu

Rajeev Balasubramonian
University of Utah
Salt Lake City, Utah
rajeev@cs.utah.edu

Abstract

Data prefetching is vital in high-performance processors and a large body of research has introduced a number of different approaches for accurate prefetching: stride detection, address correlating prefetchers, delta pattern detection, irregular pattern detection, etc. Most recently, a few works have leveraged advances in machine learning and deep neural networks to design prefetchers. These neural-inspired prefetchers observe data access patterns and develop a trained model that can then make accurate predictions for future accesses. A significant impediment to the success of these prefetchers is their high implementation cost, for both inference and training. These models cannot be trained in real-time, i.e., they have to be trained beforehand with a large benchmark suite. This results in a large model (that increases the overhead for inference), and the model can only successfully predict patterns that are similar to patterns in the training set.

In this work, we explore the potential of using spiking neural networks to learn and predict data access patterns, and specifically address deltas. While prior work has leaned on the recent success of trained artificial neural networks, we hypothesize that spiking neural networks are a better fit for real-time data prefetching. Spiking neural networks rely on the STDP algorithm to learn while performing inference - it is a low-cost and local learning algorithm that can quickly observe and react to the current stream of accesses. It is therefore possible to achieve high accuracy on previously unseen access patterns with a relatively small spiking neural network. This paper makes the case that spiking neurons and STDP offer a complexity-effective approach to leverage machine learning for data prefetching. We show that the proposed PATHFINDER prefetcher is competitive with other state-of-the-art prefetchers on a range of benchmarks. PATHFINDER can be implemented at 0.5 W and an area footprint of only 0.23 mm^2 (12 nm technology). This work shows that neural-inspired prefetching can be both practical and capable of high performance, but more innovations in SNN/STDP prefetch algorithms are required to fully maximize their potential.

ACM Reference Format:
Lin Jia, James Patrick McMahon, Sumanth Gudaparthi, Shreyas Singh, and Rajeev Balasubramonian. 2024. PATHFINDER: Practical Real-Time Learning for Data Prefetching. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/https://doi.org/10.1145/3620666.3651332

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0386-7/24/04…$15.00
https://doi.org/https://doi.org/10.1145/3620666.3651332

1 Introduction

Hardware data prefetchers have been extensively researched [3, 6, 17, 20, 31, 37, 40, 41, 52], but the memory wall continues to limit the performance of data-intensive workloads. Continued advances in data prefetching are required to improve coverage and accuracy at low overheads, especially for workloads with access patterns that are noisy or hard to predict with simple table-based or rule-based approaches.

In recent years, given the advancements in AI and deep neural networks, multiple research efforts [17, 41, 46] have explored the use of LSTMs and other neural-based techniques in predicting future data accesses. These studies have reported significant improvements over state-of-the-art prefetchers, but their high overhead means that it will take years of advancements to nudge these designs to be practical while realizing most of the benefit that has been identified.

This paper opens up an alternative path to realizing the benefits of neural-based data prefetching. We start with an implementation that is practical today and that is competitive

with, and occasionally better than, state-of-the-art prefetchers. While this work falls short of showing a large jump in prefetching performance, it provides evidence that there is value in exploring the (largely unrepresented) low overhead space of neural-based prefetching. We thus lay the foundation for future research that can continue to improve the accuracy/coverage of these prefetchers while staying within the practical scaffolding we introduce.

How is this alternative path able to achieve high-quality neural predictions at low overheads? Prior neural-based prefetchers have relied on artificial neural networks that have been remarkably successful for image and language applications. Consider the state-of-the-art neural-based delta prefetcher, Delta-LSTM [17], that utilizes an LSTM neural network to generate prefetches based on the address deltas. It performs clustering on virtual memory addresses before training to limit the vocabulary of deltas and improve the efficiency of training. Another state-of-the-art neural address correlation prefetcher is Voyager [41], which also uses an LSTM architecture. Voyager collects a memory access trace over a 50 million instruction epoch. This trace is then encoded and fed to a training algorithm to construct the model. This training requires several hours on a GPU – this is an inherent drawback of artificial neurons that rely on stochastic gradient descent, a global optimization process that requires over 10^{17} computations (for typical networks) to arrive at an accurate model. This model is then used for inference over the next 50 million instruction epoch. Both, the training and inference overheads, are many orders of magnitude higher than what is needed for a practical hardware prefetcher. The nature of artificial neurons and stochastic gradient descent are such that it is very challenging to lower the training overhead by many orders of magnitude. A pre-trained static model will not only be large, but it will also not adapt to the behaviors of new programs with new access patterns.

The key to lowering the overhead of neural-based inference and training is to adopt an alternative neuron model - a spiking neuron [13, 25] - that is inherently low-cost. Spiking neurons rely on potential increments and do not use multipliers, thus supporting many more computations per unit area than hardware for artificial neurons. They are trained on-the-fly using the Spike Timing Dependent Plasticity (STDP) algorithm. As inference is performed, each neuron adjusts its own weights based on the timing of its own inputs and outputs. Thus, training is a local process (unaffected by the behaviors of all other neurons) and can be accomplished within nano-seconds of seeing a new input pattern. The model is continuously training itself and does not require alternating epochs of training and inference. The prefetcher can thus react to new access patterns within hundreds of cycles, unlike the epoch-based technique in Delta-LSTM and Voyager. Further, STDP is a form of unsupervised learning, i.e., it does not require a pre-labeled training set. The neurons train themselves to recognize specific patterns without labels/supervision - a post-processing step then assigns labels to each neuron (a process that is easily compatible with prefetcher implementations).

As we detail in this paper, Spiking Neural Networks (SNNs) and STDP are not perfect - they do introduce a few challenges. In particular, they have historically lagged behind the accuracies of mathematically-optimal and iterative stochastic gradient descent. We make the case that our proposed prefetcher, Pathfinder, based on SNN/STDP can offer sufficiently high accuracies and can capture access patterns that are poorly handled by non-neural prefetchers. SNNs also have a time component that increases the overhead of the hardware implementation, but we show that approximate low-overhead implementations offer nearly the same accuracy. In spite of the above drawbacks, Pathfinder is implementable today, and thus represents a more practical starting point for follow-up work that can incrementally approach the higher accuracies of stochastic gradient descent.

Recent literature [6, 36] has also considered other machine learning based techniques, e.g., Pythia [6] uses reinforcement learning to identify useful deltas during run time. While such an approach is effective at run-time learning and has low implementation complexity, we observe that the random explorations inherent in reinforcement learning can cause useless prefetches and sometimes fail to identify the best delta, potentially impacting its performance.

This paper thus makes the following contributions:

- We introduce a hardware data prefetcher, Pathfinder, that is based on SNN and STDP for on-the-fly training and inference.
- We introduce novel techniques to encode inputs and label neurons to enable the SNN to learn memory access patterns similar to image recognition.
- We show that such prefetchers can be implemented at relatively low overhead and are able to predict access patterns that are problematic for other state-of-the-art non-neural prefetchers. Pathfinder can therefore be an effective piece in an ensemble of prefetchers.
- We quantitatively compare Pathfinder to a range of state-of-the-art prefetchers using the ChampSim framework. We observe that relative to these baseline prefetchers, Pathfinder is competitive on average and better on a few benchmarks. On average, Pathfinder achieves 18.7% better performance than Delta-LSTM, 9.3% higher IPC than SPP, and 2% higher IPC than Pythia. Pathfinder also exhibits 2.1% higher IPC than BO [31], a 1.7% advantage over Voyager [41], and it attains 99.12% of the effectiveness of SISB. In our optimal design configuration, the ensemble of Pathfinder, Nextline, and SISB achieves 4.6% higher IPC than BO [31], 4.1% higher IPC than Voyager [41], and surpasses SISB by 0.3%.

2 Background and Related Work

2.1 Related Work in Non-Neural Prefetchers

Much of the early work in prefetching relied on tables that tracked history, strides/offsets, delta patterns, and correlations. Given a certain access pattern, the tables are indexed to predict the next accesses in the stream. The Voyager paper refers to this class of non-neural prefetchers as rule-based prefetchers, which they categorized as strided/stream prefetchers and correlation prefetchers. Below, we briefly describe some of these major categories of prefetchers.

The simplest implementations of Strided Prefetchers are NextLine Prefetchers [21, 23, 30]. They are less effective for some data traversals [43]. Several works have predicted varying stride lengths based on recent accesses tracked in metadata tables [10, 14, 30, 38, 43]. Some works have also handled more complex streams by tracking 4 different stride lengths [19]. Delta Correlation [30, 38, 41] is a significant advancement on stride prediction. Best Offset [30] is on the simple end of this spectrum, while VLDP [40] uses a more complex implementation similar to the TAGE branch predictor [39].

A Correlation Prefetcher builds connections between past access patterns to determine future access patterns [32]. Spatial Memory Streaming [45] determines what to prefetch based on the patterns in which data was accessed and their spatial relationships [32, 45]. Bingo [4] and other spatial prefetchers [9, 22, 44, 45] also exploit similar repeating patterns. Temporal Prefetching focuses on the relative timing between block accesses. Dead-Block Predictors [27] similarly identify the last touch to a block before it gets evicted.

2.2 Related Work in Neural Prefetchers

The success of deep neural networks has inspired a few recent attempts at training deep networks to predict future memory accesses. Works like Peled et al. [36] and Pythia [6] use reinforcement learning for making predictions. Peled et al. pursue "Semantic Locality", i.e., locality defined by the algorithm or data structure. Peled et al. use the compiler to provide hints about the algorithm and the data structure being traversed. These hints are combined with information from the CPU to train a reinforcement learning model to predict future addresses. Pythia takes a somewhat similar approach in that they have a modular set of variables that can be used to train the Reinforcement Learning model. Pythia's approach also allows for customizable prefetchers and doesn't require the code to be recompiled.

Hashemi et al. [17] used LSTMs for data prefetching, and presented two different LSTM models, one that uses a large vocabulary to model all of the most common deltas, and one that uses clustering to shrink the vocabulary and the model by categorizing accesses based on locality. Srivastava et al. [46] also used LSTMs and shrank the LSTM model with a binary encoding that uses the output from multiple output neurons to determine the label. RAOP [52] further builds on this by designing an Offset delta prefetcher using LSTMs. Voyager [41] also pursues LSTMs with a hierarchical model, and divides the work of prefetching the page and offset across multiple LSTMs. Wu et al. [50] have also performed an initial study with motivations similar to ours. In a recent HotOS paper, they show that Hebbian learning principles can achieve high prefetch accuracies in the contexts of disaggregated memory systems and CPU-GPU page movement.

2.3 Why Pursue Neural Prefetchers?

Non-neural rule-based prefetchers are very effective and form the basis for most commercial prefetchers. However, data access patterns are varied and each prevailing pattern type will require a different set of rules. Capturing each of these pattern types will require a large set of rules and tables, essentially forming a "Franken-prefetcher" or an ensemble of prefetchers, that combine features from all state-of-the-art prefetchers. A neural prefetcher is a single elegant formulation that tries to capture a large number of rules as patterns that can be detected by neurons with trainable weights.

A second key advantage of a neural prefetcher is that it is tolerant to noise. For example, out-of-order execution may cause a re-ordering of loads, which in turn can impact the indices used for rule-based table structures, thus leading to wrong predictions. Noise can also be caused by variations in program control flow and interference from other co-scheduled threads or applications. Neural-based prefetchers attempt to generalize rule-based prefetchers. The hope is that if the recent history of PC/address is provided as input, the neurons will train themselves to recognize the prevailing pattern and make correct predictions even in the face of noisy inputs. Prior papers on neural prefetchers, e.g., Voyager [41], have made the case that they can achieve higher performance than rule-based prefetchers, i.e., they can make good predictions for a larger set of access patterns. Those benefits will likely be larger in noisy environments.

However, to date, a practical neural-based prefetcher does not exist - our goal is to realize the opportunity of neural prefetching at low cost.

2.4 SNN Background

Spiking Neural Networks (SNNs) attempt to emulate biological behavior [13, 25]. Most implementations approximate them with leaky-integrate-and-fire operations [13]. Each neuron has a potential; when an input spike is received, the potential is increased or decreased based on the weight for that input (integrate operation). If no input is received, the potential reduces (leak operation). When the potential reaches a threshold, it generates its own spike (fire operation). Thus, all operations are performed with adders and no multipliers are required.

Inputs are fed over many *ticks*, with 8-16 ticks typically working well [18, 25]. Inputs and outputs can be encoded in

many ways; *rate encoding* generates a number of spikes that is a function of the input value; *temporal encoding* generates a spike at a tick that is a function of the input value. Thus, it takes many ticks to process an input and generate an output sequence of spikes. This is one of the drawbacks of SNNs.

SNNs can be trained with STDP, which is a form of Hebbian Learning [7]. Each neuron adjusts its own weights without needing labeled inputs. If an input spike occured just before the output spike, that input is deemed important and its weight is incremented. If an input spike occured just after the output spike, that input is deemed unimportant and its weight is decremented. SNNs can therefore be learning constantly. In essence, an SNN starts out with random weights and when it sees an input, one of the output neurons fires. The weights are strengthened such that the same neuron firing sequence is encouraged when a similar input is seen again. Thus, the neurons in the network train themselves to recognize certain input patterns, even without requiring labels for those input patterns. As a post-processing step, we can observe each input and the corresponding firing neuron and assign an appropriate label to that output neuron. The training process can therefore converge after seeing just a few examples of each input pattern. The training process is local, i.e., during inference, each neuron can simply perform its weight adjustments based on its own behavior. This is a salient difference from artificial neural networks that require many epochs of training with a large labeled dataset, with each input being involved in forward and backward passes that reduce a global cost function.

There are many ways to implement an SNN in hardware. IBM's TrueNorth [8] and Intel's Loihi [11] are prominent commercial examples targeted at larger-scale SNNs. In addition, many academic studies have also proposed implementations [15, 24, 28, 29, 33, 34, 48]. We base our prefetcher design on the best practices introduced by these prior SNN implementations. While these implementations require a modest number of transistors to implement the logical operations, they require significantly more area/energy to store/access weights and neuron potentials across multiple ticks [8, 11].

3 The Pathfinder Architecture

This paper explores the potential of using SNNs to make prefetching decisions with high accuracy and coverage. Another primary focus is to create a design with low costs in terms of area, energy, and latency. We hypothesize that SNNs are especially compelling because they not only achieve low-cost inference, they also achieve continuous training through STDP at a very low cost. Unlike state-of-the-art neural-based solutions like Delta-LSTM and Voyager, the proposed Pathfinder is hardware-implementable and doesn't necessitate separate epochs for training and inference.

We design Pathfinder to perform online learning on access patterns within a page. When a block is accessed, it predicts the next block to be accessed within that same page. Pathfinder employs a 3-layer network alongside supporting tables used for labeling the output neurons. This section describes Pathfinder's network architecture, how inputs are encoded, how labels are computed, extensions to improve accuracy and reduce cost, and hardware design details. We end with an example showcasing the fast noise-tolerant learning that SNNs are capable of.

3.1 The Structure and Topology of Pathfinder

To keep complexity in check, we deploy a 3-layer SNN, shown in Figure 1. Our implementation consists of an input layer, an excitatory layer comprising 50 neurons, and an inhibitory layer also consisting of 50 neurons. The input layer has $D \times H$ neurons, where D is the range of allowed address deltas, and H denotes the length of delta history (more details shortly). The unsupervised learning rule, STDP, is employed to update the connections (weights) between the pre-synaptic neurons (the input layer) and the post-synaptic neurons (the excitatory output layer). The network thus has $D \times H \times 50$ weights that need to be learned.

In the subsequent evaluation, we vary the length of the delta range and the quantities of excitatory/inhibitory neurons. For now, we use the default delta range 127 (from -63 to 63) and 50 excitatory and inhibitory neurons as a reasonable starting point that captures most of the likely delta prediction outcomes. Each excitatory neuron is connected to a single inhibitory neuron; when the excitatory neuron fires, its inhibitory neuron sends inputs to all other excitatory neurons to stifle their firing. The degree of inhibition is a tunable knob that we vary to allow multiple neurons to fire, which is desirable for high-degree prefetching.

The SNN is built on well-known leaky-integrate-fire (LIF) neuron models, publicly available as part of the BindsNet [18] framework. We adopt the framework's STDP implementation to continuously update weights.

3.2 Encoding Inputs for Pathfinder

Prior work has demonstrated that SNNs are effective at various image classification tasks [13, 25]. Leveraging this experience, we aim to frame the prefetching problem as an image classification task. By adopting the right formulation, we anticipate that identifying a memory access pattern can be analogous to classifying an image, while also being resilient to some level of noise in access patterns. We therefore encode the memory access pattern as a pixel matrix, converting the H-length memory access stream into a *Memory Access Pixel Matrix* with height H and width D.

Once this conversion is done, the rest of the network can draw from best practices in image classification from prior works [13, 25], effectively enabling different neurons to fire for each detected memory access pattern. To strike the balance between coverage, accuracy, and cost, it is vital to define an appropriate set of inputs to the SNN. There is

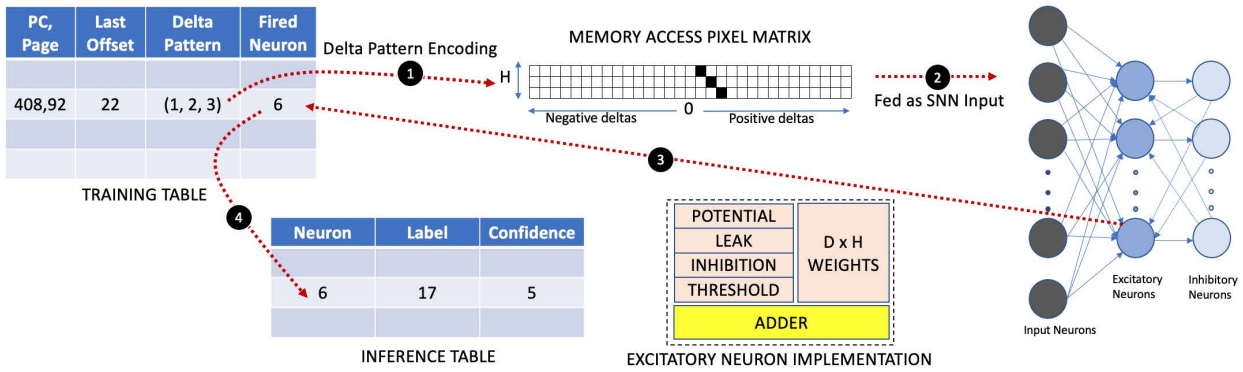

Figure 1. Example pattern being tracked and updated in PATHFINDER.

a large design space for these inputs. Our experiments and some prior works [31, 37] have empirically shown that it is effective to use per-page memory address deltas as inputs. We later also discuss and evaluate other types of inputs.

To facilitate learning, a *Training Table* keeps track of recent accesses by a given PC to a specific page (see Figure 1). For example, PC 408 is accessing page 92 and has touched blocks with page offsets 16, 17, 19, and 22. As done by many prior works [31, 37], we focus on delta patterns, i.e., the gap between consecutive addresses, which tend to be more predictable and easier to encode than the addresses themselves. Thus, in this example, the Training Table entry has a delta pattern of {1, 2, 3}. We assume that the prefetcher implements a delta history with length $H = 3$, so with a delta history of {1, 2, 3}, we are now ready to query the SNN.

❶ Our approach starts by creating a memory access pixel matrix, which transforms the delta history {1, 2, 3} into an image representation. The resulting image consists of $D \times H$ binary pixels, visually shown in Figure 1. Each row in the image represents one of the deltas from the given history, while each column signifies a different delta value. With a page size of 4KB with 64-byte blocks, the deltas range from -63 to 63, leading to $D = 127$. In Figure 1, the delta history {1, 2, 3} is illustrated with three prominent black pixels, one per row at the appropriate delta value.

❷ To feed inputs to the SNN, we use a rate coding scheme. The pixel values are converted into an input spike train of length T, following a Poisson distribution. An input interval of T ticks is used to process the SNN input.

In our initial experiments, we prioritize achieving high accuracy, and thus we choose $D = 127, T = 32$. However, we later consider ways to reduce cost by shrinking the SNN size with strategies like smaller D, smaller T, fewer neurons, etc.

3.3 Learning Labels on the Fly

The SNN operates as a black box, taking the Memory access Pixel Matrix as input and generating a set of firing output neurons. To construct the input sequences and interpret these firing output neurons, a Training table and Inference Table are employed, which we describe next.

The Training Table tracks recent combinations of PC/page and the blocks that have been touched within that page. In our ongoing example, after 4 accesses to a page, we obtain 3 deltas, which are then fed as input to the SNN. ❸ This input, along with the firing output neuron (for instance, neuron 17), is recorded in the Training Table. As depicted in Figure 1, the training table now contains an entry with delta history of {1, 2, 3}, last accessing page offset 22 (which helps calculate the future delta), and output neuron 17.

Upon encountering the same input and output pattern in subsequent instances, the Inference Table captures the next delta, let's say 6. We can now label the SNN's output neuron and assign it an initial confidence value (1 in our study). ❹ In this scenario, the output neuron 17 is associated with a label 6, which implies that when output neuron 17 fires, and its corresponding confidence is greater than 0, we initiate a prefetch for the next block in the page with a delta of 6.

Subsequent updates to the Training Table query the Training Table to identify if an output neuron label must be updated. Once a label is assigned for neuron 17, we no longer need to actively look for a label for that neuron as long as its confidence remains above 0. It is worth noting that the Training Table is an independent structure designed to understand the delta patterns with page and PC awareness. It detects consistent access patterns that the SNN has learned to recognize. Once the Training Table has detected that consistency, it assigns a label to the corresponding neuron in the Inference Table. Our label assignment process is similar to how Diehl and Cook[12] assigned labels for their model.

3.4 Additional Design Details

Confidence Estimations. In the Inference Table, each excitatory output neuron in the SNN is equipped with both a label and a confidence counter, serving as essential components for making accurate prefetching decisions. Upon every new access, we check if the prediction generated by

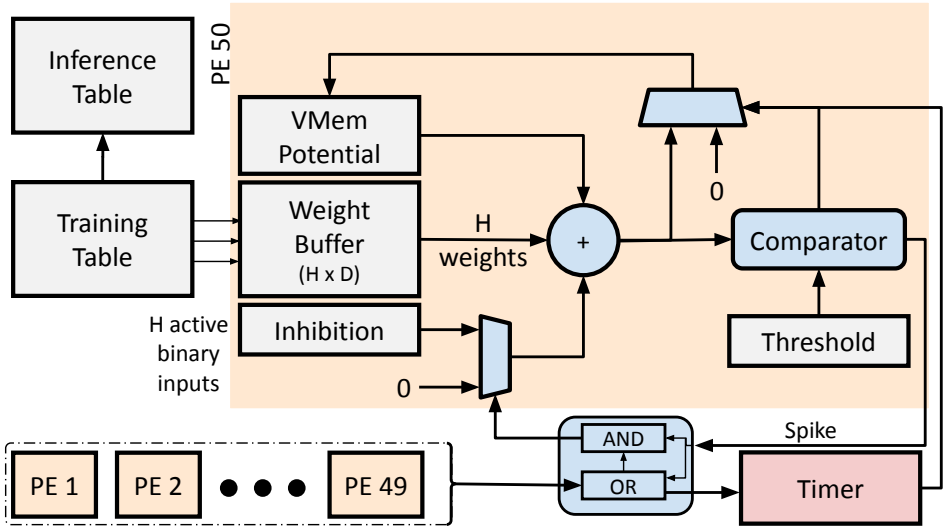

Figure 2. Circuit blocks in one PE (one neuron) of PATHFINDER.

the SNN for its prior access aligns with the accessed block of the current page. We first use the page and pc information to index the Training Table to locate the fired neuron that got activated. Then, we use the neuron's index to get its associated label in the Inference Table, thus serving as the prediction. When the SNN's prediction matches the accessed block of the current page, the confidence associated with the corresponding neuron is incremented. Conversely, if the prediction does not match the accessed block, the corresponding neuron's confidence is decremented. The confidence is implemented as a 3-bit saturating counter. If the confidence reaches zero, the neuron's label is erased. This re-initiates the process of tracking a specific delta pattern and finding its corresponding label. Thus, not only does the SNN learn continuously with STDP, but the confidence estimations also help clear labels and adapt to new patterns as the program moves between phases.

Multi-Degree Prefetching. In our previous discussion, each neuron is associated with a single label, resulting in a prefetcher with a degree of 1. Most state-of-the-art prefetchers have the ability to increase coverage and also IPC by prefetching with higher degrees. To get better IPC and coverage, we recognize the need for PATHFINDER variants capable of supporting higher degrees of prefetching.

To address the above challenge, we can vary the inhibition degree to control the number of excitatory neurons that fire for a given input. By reducing the inhibition, we can observe 2-5 firing excitatory neurons for any input. We can then issue prefetches for the firing neurons with confidence values exceeding a certain threshold or select the neurons with the highest confidence levels.

Alternatively, we can use high inhibition to allow only one firing neuron, but assign multiple labels (and confidence values) to that single firing neuron. For each neuron in the Inference Table, we accommodate two slots for the label-confidence pair. For instance, when neuron 17 fires upon observing delta pattern {1, 2, 3}, it obtains its label following the process described above. Later, if it fires when encountering delta pattern 2, 2, 3 with the next access delta being 12, it will get a different label 12 to be assigned to it. Therefore, neuron 17 will have the two labels, 6 and 12, attached to it when it identifies a different pattern. This approach has been adopted in our work to support higher degrees of prefetching to enhance the performance and efficiency of PATHFINDER.

Ensemble of Prefetchers. Another avenue to increase prefetching coverage is ensembling PATHFINDER with another state-of-the-art, low-overhead prefetcher. In scenarios where PATHFINDER may not generate sufficient high-confidence prefetches, the prefetching budget can be allocated to prefetches identified by other prefetchers. We earlier claimed that PATHFINDER is effective at generalizing several rule-based prefetchers - but here, we are falling back on rule-based prefetchers. It turns out that PATHFINDER is quite selective in issuing prefetches - it waits to see the same pattern multiple times and needs high-confidence labels. The other rule-based prefetchers are effective when the patterns are simple and can be triggered without much deliberation, such as a next-line prefetcher. Therefore, in our evaluations, we combine PATHFINDER with a state-of-the-art prefetcher like an idealized version (SISB) of the Irregular Stream Buffer [20], as well as with one of the simplest prefetchers like Next-Line to enhance prefetching performance.

Initial Accesses to a Page. As mentioned earlier, PATHFINDER currently only focuses on predicting the next block touched within a page. Predicting the first access to a page that has not been touched in a while (a cold page access) is left for future work. The design so far presents a limitation wherein the SNN is fed with the last H deltas, and prefetching is

initiated only after the first H+1 accesses to a page. With $H = 3$ for most of our experiments, it is a significant missed opportunity to delay prefetch until the 4th access to a page.

We propose an extension to the input encoding scheme, handling the first three accesses to a page as special cases, to address the above limitation. Upon the initial touch to a page where the page is absent in the Training Table, the offset is OF1. The SNN receives a delta input of {OF1, 0, 0}. On the second touch to the page, with a single delta D1 available, the SNN receives a delta history of {0, 0, D1}. Note that the zeroes have moved from the end to the start of the history so the SNN can make a distinction between an offset pattern and a delta pattern. For the third touch, we feed the two available deltas ($D1$ and $D2$) as {0, $D1$, $D2$}. With this extension, we can initiate inputs to the SNN whenever any access to a page is observed, eliminating the necessity to wait for 3 deltas to construct the Input Matrix.

Enlarged Pixel in Input Pixel Matrix. We encountered a challenge with the Input Pixel Matrix being extremely sparse, which posed difficulties in triggering neuron firing. To alleviate this issue, we amplify the signal by expanding each colored pixel so it also occupies its 4 neighboring pixels. This amplification technique empirically demonstrated noteworthy benefits, such as facilitating the recognition of patterns, elevating the neuron firing rate, and ultimately leading to improved IPC. However, a potential issue emerged after signal amplification, where some closer pixels clustered together, resulting in aliasing problems. To mitigate this concern, we shift the middle delta in the delta pattern by a fixed constant, effectively reducing the risk of aliasing.

Lowering Time Interval. A drawback of SNN inference is that inputs are processed over a T-tick interval, which adds to the prefetcher's latency and energy consumption. Ideally, we'd like to compress the input time interval to one and still produce the same prediction. To approximate the behavior of a 32-tick SNN with a low-cost implementation, we model a 1-tick SNN and assume that the neuron with the highest potential after 1 tick would have been the first to fire. Our empirical findings, shown in Table 1, demonstrate a large percentage of the highest voltage neurons after the first tick are also the most-firing neurons during a 32-tick interval. This correlation indicates our 1-tick approximation closely aligns with the SNN's behavior during the 32-tick interval, offering high accuracy at a low implementation cost.

3.5 Implementing PATHFINDER in Hardware

PATHFINDER is composed of an SNN and two supporting tables that help with encoding and labeling. Figure 2 shows details of the PATHFINDER's microarchitecture. The SNN itself is the largest overhead due to the storage requirements for its weights. The SNN is primarily composed of 50 excitatory neurons, each equipped with DH weights. Each of these 50 neurons is implemented with two adders and a directed

Benchmark Suite	Trace Name	matched neuron
GAP	cc-5	92.43%
GAP	bfs-10	90.63%
SPEC06	471-omnetpp-s1	93.56%
SPEC06	473-astar-s1	91.38%
SPEC06	450-soplex-s0	85.72%
SPEC06	482-sphinx-s0	86.21%
SPEC17	605-mcf-s1	91.92%
SPEC17	623-xalan-s1	93%
Cloudsuite	cassandra-phase0-core0	86.59%
Cloudsuite	cloud9-phase0-core0	82.76%
Cloudsuite	nutch-phase0-core0	85.66%

Table 1. The % of neurons with the highest voltage after the first tick that matched the firing neuron after 32 ticks.

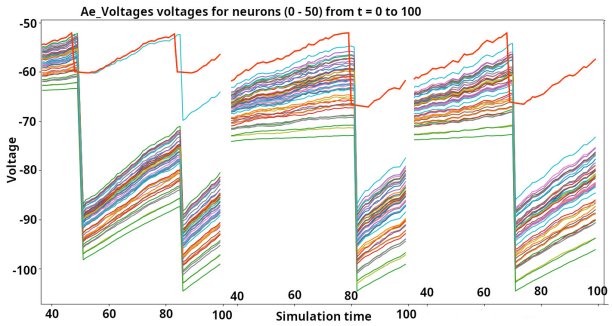

Figure 3. Demonstration of SNN learning a pattern. Neuron 9, shown in orange, trains itself to recognize an input pattern that is fed repeatedly to the SNN for three input intervals.

mapped scratchpad with a little more than DH entries to track weights, potential, leak, inhibition, and firing threshold.

The sequence from the Training Table is converted into a set of spikes, which are fed as a set of spike timestamps/ids. These ids are used to sequentially access weights from the weight buffer, which are then fed to the ALU to increment neuron potentials (or decrement based on leak). Also given the input process interval and the high sparsity, an output spike can be generated after tens of ALU increments. Given the low cost of an ALU, parallel ALUs can implement these increments at low overhead. This helps to lower the latency of a prefetch prediction to just a handful of cycles.

The supporting tables are relatively small but are complicated by the required indexing mechanisms. The Training Table is indexed with a combination of PC/page to track deltas and identify labels. It is therefore implemented as a CAM structure. Based on our analysis, a Training table with 1K 120 bit-wide rows is good enough to maintain high performance. We estimate area under 0.02 mm^2 and power under 11 mW using CACTI [5] to model the structures at 22 nm and then scaled down to 12 nm. The firing neuron indexes the Inference Table, and is also implemented as a CAM. The Inference Table has 50 rows, each with 24 bits, yielding an area of 0.00006 mm^2 and power of 0.02 mW However, many

of these parameters can be further reduced with minimal impact on prediction accuracy, as we show later in Section 5.

To more precisely estimate its overhead, we implemented and functionally verified the 50-neuron SNN shown in Figure 2. The SNN is modeled with 50 PEs (each with its adder, comparators, and weight buffer) and a global timer module. The spikes generated from each neuron are aggregated as a single signal and then used to inhibit neurons that do not fire. We use the Synopsys Design Compiler [47] at a 12-nm technology node to determine the power and area consumed by this structure. The SNN uses 0.21 mm^2 of area while consuming a peak power of 446 mW, with the weight buffer accounting for 56% of area and 94% of power. These metrics are at a clock frequency of 1 GHz, and the circuit can operate correctly up to a frequency of 2.5 GHz. The hardware overhead of Pathfinder is comparable to that of Pythia [6], which at 14nm has a 0.33 mm^2 area overhead and a power consumption of 55.11 mW. Pythia has a larger storage requirement, which results in the larger area. Pathfinder implements the weight buffers as register files, which accounts for its larger power profile, relative to Pythia. A modern processor at 12 nm technology, the AMD Ryzen 7 2700X [49], has a die size of 213 mm^2 and TDP of 105 W - making the overheads of Pathfinder a small fraction (< 1%) of the total processor.

3.6 SNN in Action

We now use a few representative examples to zoom into the inner workings of an SNN to show how it quickly learns new patterns with little effort and tolerates noise. Consider an input pattern of {1, 2, 4} fed to an SNN initialized with random weights. Our example walks through access patterns listed in Table 2, accompanied by the visualization depicted in Figure 3.

When the input pattern is first fed to the SNN, one of the excitatory neurons in our network - neuron 9 in our example - fires at tick 42. Other neurons come close to firing (with potentials slightly below the firing threshold), but they are subsequently inhibited and do not fire within the 100-tick input interval. The left part of Figure 3 shows that the top orange neuron 9 fires multiple times during the first input interval. This is a crucial element in unsupervised learning - without any labels and without any training, neuron 9 is pre-disposed to detecting the pattern {1, 2, 4}. Neuron weights are then adjusted to increase neuron 9's ability to detect this input pattern. When pattern {1, 2, 4} is fed to the SNN again in the next input interval, neuron 9 continues to fire again. With a single observation and STDP weight adjustment, neuron 9 has better trained itself to recognize this pattern than the other neurons. We see this in Figure 3 – the potential of the orange neuron 9 has further distanced itself from the potentials of other neurons in the 2nd and 3rd input intervals.

Input access pattern for the SNN	Firing neuron	Firing tick	Potential of the next-best neuron
{1, 2, 4}	9	42	-52.3
{1, 2, 4}	9	65	-51.9
{1, 2, 4}	9	65	-51.9
{1, 2, 4}	9	52	-51.9
{1, 2, 4}	9	54	-51.7
{1, 2, 4}	9	45	-51.7
{1, 3, 4}	24	70	-51.6
{1, 2, 5}	9	63	-51.6
{1, 4, 2}	49	77	-52.2
{1, 3, 6}	0	80	-52.17
{1, 2, 4}	9	46	-51.4

Table 2. Understanding the SNN's firing/learning behavior.

Table 2 summarizes our observations for a set of SNN input patterns. It shows when pattern {1, 2, 4} is fed repeatedly, neuron 9 fires each time and after the initial learning, it detects the pattern at earlier ticks. We also see that the potential of the next nearest neuron starts to fall behind because of the STDP weight adjustments and lateral inhibition. Table 2 also shows what happens when the input pattern is slightly changed. We show three input patterns that are slightly different from {1, 2, 4}, and in one of them, neuron 9 recognizes the pattern in spite of the noise. Note that this demonstration is only showing the SNN's ability to recognize patterns - neuron labeling is a separate step not shown here, and is done once we identify the next delta after 1, 2, 4.

4 Methodology

Pathfinder is a neural prefetcher that facilitates on-the-fly training and inference by swiftly adapting the predictions in response to evolving program phases. We compared it against a range of state-of-the-art prefetchers emphasizing different perspectives, including neural prefetchers and rule-based prefetchers that learn delta patterns and address correlations on a set of benchmarks from GAP, SPEC06, SPEC17, and Cloudsuite.

4.1 Simulator

To evaluate our proposals, we used a fork of ChampSim developed by the authors of the Voyager paper [41] to conduct our single-thread simulations. This fork was specifically designed for the ML-based Data Prefetching Competition [2] and used to test several state-of-the-art ML-based prefethers[41, 51]. The key distinction of this fork from the main ChampSim branch is its ability to accept precomputed prefetch traces. First, the memory trace is utilized to generate a prefetch file through prefetching techniques. Second, both the original memory trace and the generated prefetch trace are fed into ChampSim to have cycle-accurate simulation and IPC estimates. ChampSim's memory hierarchy simulation parameters are concisely summarized in Table 3. Similar

to the setting in the ML Prefetching Competition, we focus on LLC misses and prefetching from memory to the LLC.

L1I	32KB, 64 sets, 8 ways, latency 4 cycles
L1D	48KB, 64 sets, 12 ways, latency 5 cycles
L2	512KB, 1024 sets, 8 ways, latency 10 cycles
LLC	2MB, 2048 sets, 16 ways, latency 20 cycles
DRAM	tRP = TRCD = tCAS = 12.5 1 channel, 8 ranks per channel, 8 banks per rank Write Queue Size = Read Queue Size = 64

Table 3. ChampSim parameters.

4.2 Pathfinder

Pathfinder uses an off-the-shelf SNN simulator to make prefetch predictions. The topology of the SNN is implemented using a PyTorch-based [35] library, Bindsnet [18], for a fast GPU-enabled SNN simulation. Within BindsNet, we deployed the provided "DiehlAndCook" model as our baseline and modified it to handle a different input shape (our $D \times H$ Memory Access Pixel Matrix instead of an MNIST image). Other SNN parameters are summarized in Table 4. We used the built-in BindsNet monitor classes to measure run-time neuron behaviors. Input spike trains were generated using the Poisson rate encoding included in BindsNet.

n_input	D × H; D=128, H=3
n_neurons	50
exc	20.5
inh	17.5
dt	1
norm	38.4
theta_plus	0.05
inpt_shape	(1, 128×3)
number of time steps (ticks)	32

Table 4. BindsNet network initialization parameters.

4.3 Baselines

Delta-LSTM [16] is an LSTM-based neural prefetcher that predicts virtual memory address deltas. As suggested in the paper [16], we implement a 3-layer structure - two LSTM layers with 128 neurons each and one dense layer. To reduce training overhead and achieve better performance, we followed the paper's recommendation to cluster each trace file into 6 clusters based on the locality of memory addresses and performed training and testing runs on these 6 clusters separately. In the current implementation, training is performed on the initial 10% of accesses in each cluster, while inference is performed on the full trace.

Voyager [41] is a neural-based prefetcher that learns address correlations using an LSTM architecture. For the Voyager code, we use the publicly available version on their Github [42]. Voyager envisioned an implementation where traces collected during a 50 M instruction epoch are trained offline, then deployed for the next 50 M instruction epoch. To ensure all the prefetchers use the same trace files, we trained and tested Voyager on the same trace files we use for all the baseline prefetchers and Pathfinder.

SPP [26] is a history-based delta prefetcher. The prefetch degree is dynamically adjusted through adaptive throttling, allowing SPP to issue a prediction only when its confidence level surpasses a predetermined threshold. Our comparison against SPP uses the ML-based Champsim simulator in [51], identical to our own simulator, with an integrated SPP implementation.

Pythia [6] is a state-of-the-art prefetcher that applies reinforcement learning to data prefetching for the L2 cache. Pythia's model is built around actions that are delta prefetches, and state that includes a dynamic range of "features" like the addresses, program counter, and it uses performance counters for bandwidth or IPC to help compute rewards. Using the code in Pythia's public repository, we ported Pythia to serve as a prefetcher into the LLC, and additionally tested across several diverse configurations that primarily varied the action list and the alpha, gamma, and epsilon values to identify the best-performing configuration at the LLC.

More Baselines In our evaluation, we also included other baseline approaches for comparison: a baseline with no prefetching, and two baseline prefetchers integrated into the ChampSim fork - Best Offset (BO) [31] (with prefetch throttling disabled by the provider) and an idealized version (SISB) of Irregular Stream Buffer [20], which are both provided by the ML-based Data Prefetching Competition [2].

4.4 Benchmarks

Our evaluation includes programs drawn from various benchmarks: GAP, SPEC06, SPEC17, and CloudSuite. The traces for these benchmarks were provided by the ML Prefetching Competition [2] and the 2nd Cache Replacement Championship [1]. Our simulations for each benchmark consist of 1 million memory loads, equivalent to tens of millions of benchmark instructions, as summarized in Table 5.

Before each simulation, we performed a warm-up phase of 10 million instructions to prime the caches. Since Pathfinder learns online and exhibits rapid pattern recognition, we omitted a separate training phase for Pathfinder compared to the baseline prefetchers, and our results reflect the behavior of the prefetcher when processing a burst of 1 million memory accesses.

4.5 Metrics

The primary evaluation metrics are: IPC, issued prefetches, accuracy, and coverage. All simulations execute the same subset of the trace file to ensure a fair comparison. We computed

Benchmark Suite	Trace Name	Total instructions
GAP	cc-5	31M
GAP	bfs-10	71M
SPEC06	471-omnetpp-s1	65M
SPEC06	473-astar-s1	99M
SPEC06	450-soplex-s0	39M
SPEC06	482-sphinx-s0	95M
SPEC17	605-mcf-s1	48M
SPEC17	623-xalan-s1	63M
CloudSuite	cassandra-phase0-core0	207M
CloudSuite	cloud9-phase0-core0	208M
CloudSuite	nutch-phase0-core0	154M

Table 5. Tested workloads.

accuracy by dividing the reported useful prefetches by the total number of issued prefetches. We computed the coverage by dividing the useful prefetches by the number of baseline misses. All prefetchers submit at most 2 prefetches for each memory access, so the total number of issued prefetches is at most twice the number of the baseline misses.

5 Results

Prior work [17, 41, 46, 52] has shown that neural prefetchers like Voyager can outperform rule-based prefetchers. Our evaluation assesses if Pathfinder, an implementable neural prefetcher, can approach or exceed the performance of hard-to-implement neural-based delta prefetchers like Delta-LSTM and neural prefetcher Voyager that learns address correlations. We further assess if Pathfinder is competitive with reinforcement-learning-based delta prefetcher, Pythia, and other rule-based prefetchers. Our results show that this is indeed the case, and follow-up work must continue to explore the design space of combined neural and rule-based prefetchers that capture the most common data access patterns at low implementation costs. We start by evaluating a version of Pathfinder that is designed for high accuracy - we then show the sensitivity of Pathfinder as parameters are varied to achieve even lower implementation costs.

Figure 4a compares the IPC achieved by various prefetchers, including the no-prefetching baseline, rule-based prefetcher (BO), temporal prefetcher (SISB), neural prefetcher that learns address correlations (Voyager), delta neural prefetcher (Delta-LSTM), and reinforcement-learning-based delta prefetcher (Pythia), Pathfinder, and Pathfinder combined with next-line (NL) prefetching and SISB. On average, we see that the selective Pathfinder offers better IPC than rule-based BO, and neural prefetchers Delta-LSTM and Voyager, and being competitive with RL-based Pythia and temporal prefetcher SISB. Furthermore, Pathfinder shows higher accuracy (Figure 4b) and coverage (Figure 4c) than all the baseline prefetchers on average besides SPP on accuracy. SPP is selective in the high-confidence prefetches that it issues, giving it the highest accuracy, but also lower coverage than other prefetchers (see Table 6). The Delta-LSTM model is not as competitive as the other prefetchers, primarily because it is trained on a subset of the trace and it encounters several new deltas during testing. We conducted an experiment where we trained the Delta-LSTM model on 30% of the full trace – the experiment with 30% training data reduced the number of unseen deltas by 10.3%, accompanied by a 4.6% improvement in IPC.

Trace	**SPP**	**Pythia**	Pathfinder
cc-5	116k	1.95M	1.865M
605-mcf-s1	642k	1.95M	1.43k
bfs-10	935k	1.73M	1.9297M
450-soplex-s0	598k	1.84M	1.718M
623-xalan-s1	1.41M	1.60M	1.985M
471-omnetpp-s1	944k	1.524M	1.8887M
482-sphinx-s0	938k	1.944M	1.965M
473-astar-s1	286k	1.94M	1.4599M
Cassandra	1.15M	2.364M	1.7535M
Cloud 9	602k	1.79M	1.767M
Nutch	892k	1.9075M	1.468M
average	774k	1.867M	1.75M

Table 6. Issued prefetches of SPP (the baseline with lowest coverage), Pythia (the baseline with highest coverage), and Pathfinder.

On most workloads, Pathfinder and Pythia exhibit similar IPCs, although with very different techniques to learn deltas. The key difference is: Pythia randomly explores different deltas so there are opportunities to learn via reinforcement, whereas Pathfinder's neurons detect patterns with high confidence. Therefore, Pythia is a more aggressive prefetcher (see Table 6) and tends to have higher coverage. This gives Pythia the edge on a few workloads. Notably, on mcf, an irregular workload, Pathfinder does worse than Pythia because it doesn't encounter as many high-confidence predictions. However, Pathfinder can bridge the coverage gap by combining with other simple prefetchers as part of an ensemble. On the other hand, the random exploration of deltas in Pythia can limit its effectiveness in some cases, e.g., consuming memory bandwidth to learn hard-to-predict patterns. We observed that Pythia's reinforcement learning can settle on a "local minimum", e.g., quickly settling on a delta of 1 in xalan, whereas Pathfinder achieves higher IPC because it identifies other better-performing deltas. Additionally, Pythia has a large set of configurations that must be tuned for high performance, whereas Pathfinder is able to quickly learn most patterns without much fine tuning.

It is more insightful to examine behaviors on individual benchmarks. Programs like xalan, soplex, omnetpp, and sphinx3 do better with SISB's temporal recording-replaying approach, which significantly outperforms the neural-based approach of Voyager and Pathfinder. However, an ensemble that combines Pathfinder and NL or SISB is able to alleviate most of this drawback. Note that our ensemble

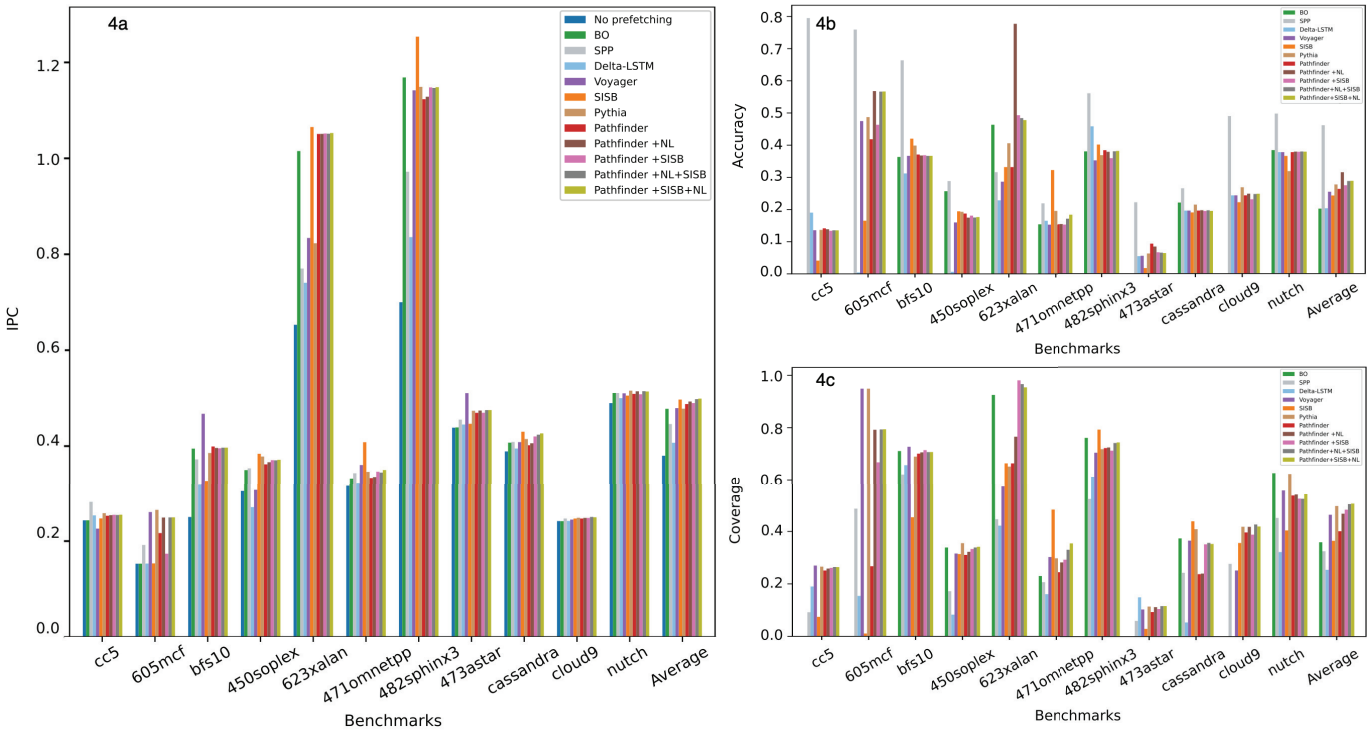

Figure 4. Performance (IPC, Accuracy, Coverage) comparison of different prefetchers on 1M load accesses. PATHFINDER configuration: 50 neurons with 2 labels for each neuron, delta range: -63 to 63, input interval: 32 ticks, prefetch degree: 2.

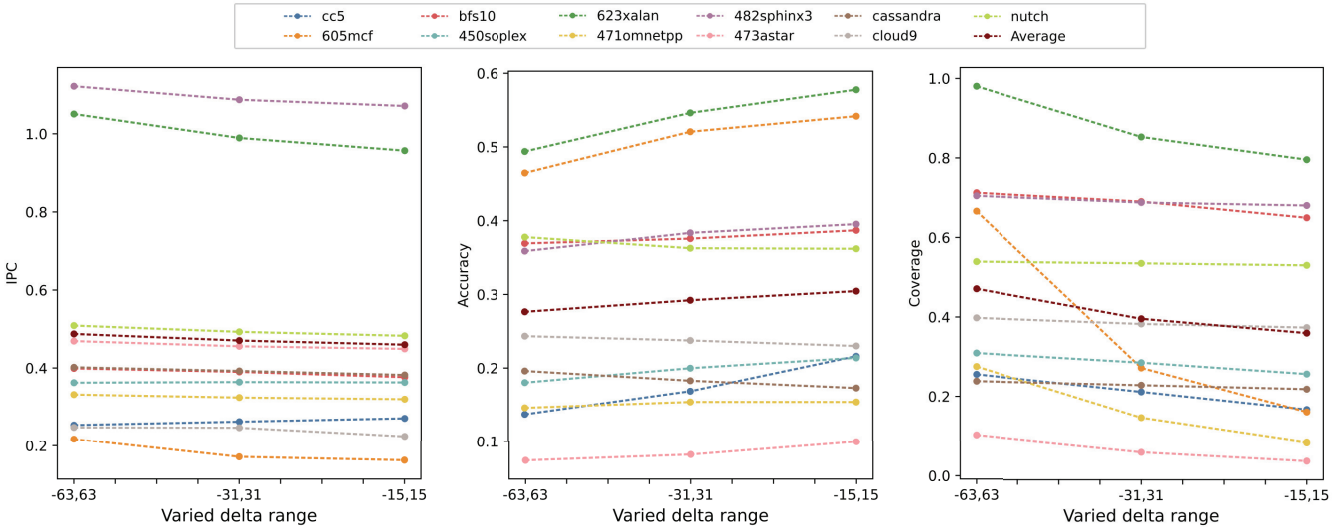

Figure 5. Performance of PATHFINDER with different delta ranges (31, 63, 127), but same neuron count (50) and time interval (32 ticks).

prefetcher prioritizes PATHFINDER and uses NL or SISB predictions to fill unused slots, so it falls a little short of SISB performance in some benchmarks. This points to an avenue for future work that explores different ensemble policies.

However, benchmarks astar, mcf, bfs, and cc benefit significantly from the neural-based approach. In all of these programs, Voyager and PATHFINDER outperform SISB, showing the potential of a neural-based approach. In 3 of these 4 programs, PATHFINDER is not as effective as Voyager - this is

to be expected given that Pathfinder is learning on-the-fly with STDP while Voyager has the benefit of a long and precise training process on the entire trace. Bridging this gap with better learning policies is a key area for future work. The gap is especially wide for mcf, but with the help of the no-overhead NL or temporal prefetcher SISB (maintaining prefetch degree=2), Pathfinder achieves performance very close to Voyager and surpasses all baseline prefetchers in accuracy and coverage. This highlights that Pathfinder is a selective prefetcher and can find distinct patterns better than other prefetchers. It is worth noting that Voyager generally lags behind Pathfinder in accuracy on most benchmarks (cc, bfs, soplex, xalan, omnetpp, sphinx, and astar). This indicates that fast learning and adaptation to short-lived patterns with Pathfinder's STDP can be more effective than capturing an average pattern across a large epoch.

The best design point is a combination of NL, SISB and Pathfinder, an implementation that benefits from both temporal-based and neural-based prefetching, which prioritizes Pathfinder's predictions and uses the prefetches from NL and SISB to fill out the remaining available slots. For our benchmarks, the neural prediction is used 80-99% of the time and the rule-based prediction is used 1-20% of the time. It is possible to get larger benefits with dynamic ensemble priority policies. Because of our fixed priorities, the ensemble can sometimes behave very similar to Pathfinder, which in some benchmarks is worse than SISB-only.

Benchmarks	#deltas in (-31,31)	#deltas in (-15,15)
cc-5	682K	425K
bfs-10	968K	880K
471-omnetpp-s1	539K	314K
473-astar-s1	665K	351K
450-soplex-s0	844K	713K
482-sphinx-s0	930K	891K
605-mcf-s1	709K	476K
623-xalan-s1	809K	712K
cassandra-phase0-core0	793K	513K
cloud9-phase0-core0	893K	721K
nutch-phase0-core0	651K	529K

Table 7. Number of deltas out of 1M load memory accesses under different delta ranges.

Next, we explore various implementations of Pathfinder to understand its sensitivity to SNN parameters. The advantage of Pathfinder lies in its low cost, so we aim to reduce its size while maintaining performance values similar to those shown in Figure 4. In Figures, 5, 6, 7, 8, we vary one parameter at a time - the delta range, the number of excitatory/inhibitory neurons, the input interval size, and training period. Finally, Figure 9 illustrates the collective impact of all these different implementations of Pathfinder.

Among the variables we considered, the length of the delta range directly determines the size of the input layer. As the delta range decreases, a tradeoff between accuracy and coverage arises, impacting the IPC (Figure 5). Smaller delta ranges reduce coverage for all programs since fewer deltas are within the shorter range (see Table 7). On the other hand, the accuracy of all programs increases because the smaller range filters out large deltas that contribute to offset predictions. Specifically, for programs like xalan and mcf, a clear drop in IPC can be observed.

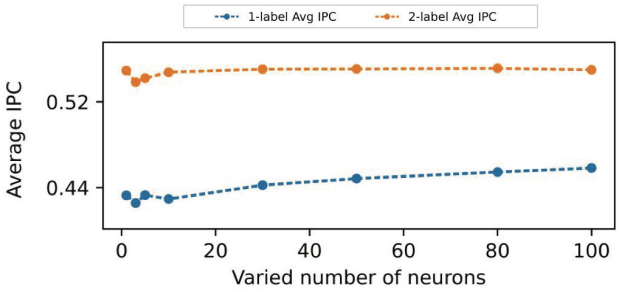

Figure 6. Comparing Pathfinder's performance with varying numbers of neurons (10-100) across two configurations: 2 labels per neuron and 1 label per neuron. Y-axis range adjusted for clarity.

Benchmarks	Avg # deltas	Avg # distinct deltas	Sum of occurrences of top5 distinct deltas
cc-5	377	182	106
bfs-10	920	329	356
471-omnetpp-s1	67	6	52
473-astar-s1	26	7	19
450-soplex-s0	775	290	355
482-sphinx-s0	842	35	645
605-mcf-s1	35	25	8
623-xalan-s1	451	14	264
cassandra-phase0-core0	163	97	135
cloud9-phase0-core0	528	269	317
nutch-phase0-core0	615	152	529

Table 8. Number of deltas, distinct deltas, and the summation of top 5 frequent distinct deltas out of every 1K accesses.

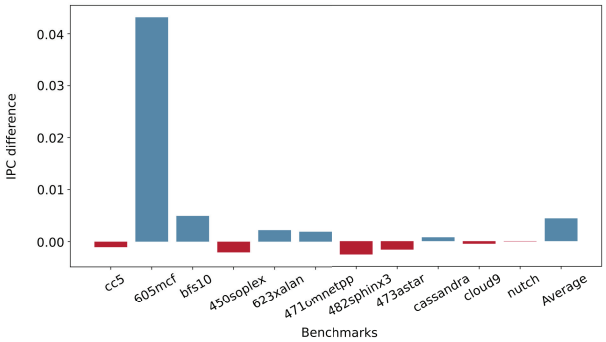

Figure 7. IPC improvement of 1-tick over 32-tick version.

Another key parameter in Pathfinder is the number of excitatory/inhibitory neurons, which significantly impacts its overall cost as it determines the size of the weight buffer and

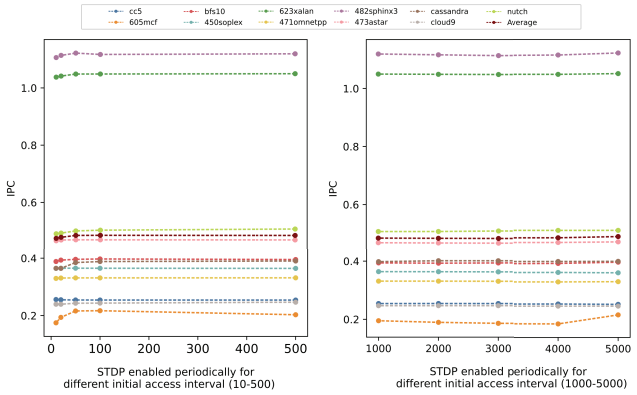

Figure 8. Performance comparison for PATHFINDER with STDP fully enabled throughout and with STDP enabled periodically for different initial accesses (e.g., on for the initial 10, 20, 50, ... accesses) in every 5K access epoch.

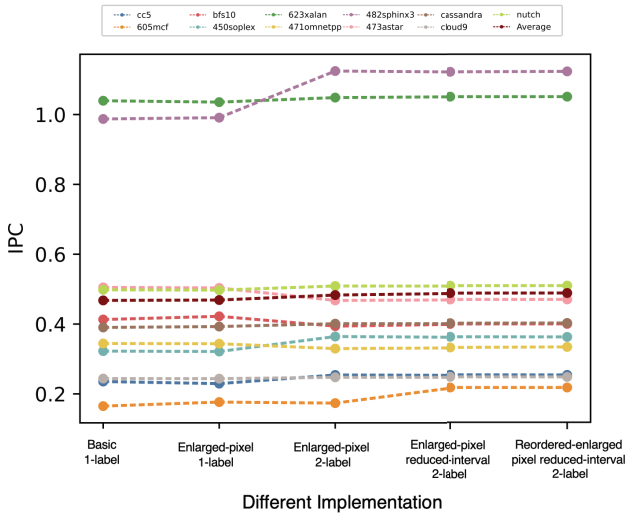

Figure 9. Comparing the performance of various PATHFINDER variants: basic 1-label version, enlarged-pixel 1-label version, enlarged-pixel 2-label version, enlarged-pixel reduced-interval 2-label version, and reordered-enlarged-pixel reduced-interval 2-label version.

Prediction Table. As previously discussed, PATHFINDER can assign multiple labels to a single neuron. We conduct an analysis to explore the influence of different neuron configurations on two PATHFINDER implementations: the PATHFINDER 2-label version and PATHFINDER 1-label version.

In Figure 6, we show IPC as the number of neurons changes for both settings. We observe that altering the neuron count has minimal impact on the 2-label version, while it more noticeably reduces the IPC in the 1-label version. To understand this better, we quantify relevant statistics in Table 8. Within a specific period, such as every 1000 accesses, the number of distinct deltas is not excessively large, with a few deltas being repeatedly observed. This indicates that PATHFINDER is agile in effectively learning short-lived patterns and adjusting its labels as the workload changes. The agility is higher when each neuron is associated with 2 labels.

Table 9 quantifies the area and power consumption for each low-cost implementation - we see that reducing the delta range and number of neurons are the most effective ways to reduce the cost.

Moreover, as discussed in Section 3, we can reduce the input interval to reduce latency and cost. Figure 7 supports that hypothesis. The IPC is affected by very small amounts, showing that the neuron with the highest voltage after the first tick of processing has a dominant likelihood of being the neuron that eventually fires during the full 32-tick interval.

In addition to reducing the size of PATHFINDER's structure, we also explore the impact of periodically disabling STDP to reduce the cost. By disabling STDP, we can save energy by not updating weights in weight buffers. Figure 8 illustrates that PATHFINDER can learn delta patterns rapidly and accurately. In our experiments, we apply STDP for only the first 10, 20, 50, 100, 1000, 2000, 3000, or 4000 accesses out of every 5000 accesses. After this initial phase, STDP is turned off with no weight updates during prediction for the remaining accesses in that epoch. The result in Figure 8a indicates that having STDP active for approximately the first 50 accesses out of every 5000 accesses is sufficient for PATHFINDER to learn the patterns and generate accurate predictions, similar to the performance achieved with STDP fully enabled.

Different Parameters	Total Area (mm^2)	Total Power (W)
50 pe, range 127	0.21	0.446
50 pe, range 63	0.107	0.227
50 pe, range 31	0.055	0.116
1 pe, range 127	0.004	0.009
1 pe, range 63	0.003	0.006
1 pe, range 31	0.001	0.002

Table 9. Area and power of PATHFINDER implementations.

Figure 9 presents major PATHFINDER implementation variants, where we begin with the basic configuration in which each excitatory neuron is assigned only one label. This already achieves competitive IPC similar to baseline prefetchers. Building upon this foundation, we explore further enhancements to boost PATHFINDER's performance. First, we enlarge the input pixels to increase the chances of neuron firing, thereby enabling PATHFINDER to capture more patterns. Second, we introduce the assignment of two labels to each neuron, which offers greater flexibility and learning capacity for the network. Third, we reduce the input process time interval, enabling PATHFINDER to adapt and respond rapidly to changing data access patterns. Last, we reorder the input pixels, which aids in optimizing the processing flow and improving PATHFINDER's overall performance.

The implementation details for these modifications are further elaborated in Section 3. By incorporating these enhancements, PATHFINDER demonstrates improved prefetching capabilities, making it a highly competitive and efficient solution for accurately predicting data access patterns.

6 Conclusions

This paper introduces an implementable path towards highly accurate neural-based prefetching that learns delta patterns at run time. Pathfinder relies on SNNs and STDP to learn data access patterns within hundreds of cycles, and uses a few supporting tables to track histories, assign labels, and maintain high-confidence predictions. We show that Pathfinder can be implemented within area and power footprints of 0.23 mm^2 and 0.5 W. Our analysis of IPC, accuracy, and coverage shows that Pathfinder is competitive with other neural and non-neural state-of-the-art that focus on learning delta access patterns and address correlations - the averages are similar, with each prefetcher yielding best results on part of the benchmark suite. Pathfinder can be combined with NL and SISB to slightly edge out the competition on average. We observe that Pathfinder is selective in its predictions, thus yielding high accuracy; we also observe that it achieves high accuracy when the implementation is scaled down. The paper identifies several avenues for future work that can realize the potential of delta neural prefetching while retaining low implementation overheads.

Acknowledgements

We appreciate the anonymous reviewers for their invaluable feedback, which improved the quality of this work. Additionally, we thank our lab mates for many insightful discussions and to Ian Lavin for his contributions. This work was supported in parts by NSF grants CCF 2224463, CCF 2217154, CNS 2245999, and Intel.

A Artifact Appendix

A.1 Abstract

The artifact appendix includes the GitHub links of prefetcher code bases, datasets info, and scripts to reproduce Pathfinder and generate IPC, accuracy, and coverage results from Pathfinder and the baselines (No Prefetch, Best Offset, SISB, SPP, Voyager, and Pythia).

A.2 Artifact check-list (meta-information)

- **Dataset:** Will be downloaded using the provided script.
- **Run-time environment:** Create a conda environment using environment.yml that is provided in the GitHub link.
- **Hardware:** Most modern general-purpose CPUs should work.
- **Metrics:** IPC, prefetcher's Coverage, Accuracy
- **Experiments:** Use the provided script to download the datasets, run Pathfinder, and generate result files that contain IPC numbers, LLC LOAD ACCESS, and LLC PREFETCH REQUEST, ISSUED to calculate accuracy and coverage numbers.
- **How much disk space required (approximately)?:** 20 GB
- **How much time is needed to prepare workflow (approximately)?:** 2 hours includes downloading the datasets
- **How much time is needed to complete experiments (approximately)?:** About 40 hours for Pathfinder
- **Publicly available?:** Yes. https://github.com/linjiaty/Pathfinder.git

A.3 Description

A.3.1 How to access. The source code for Pathfinder can be found at https://github.com/linjiaty/Pathfinder.git.

A.3.2 Hardware dependencies. Most general-purpose CPUs should be good to run this code. Baseline prefetchers may need to run with a GPU.

A.3.3 Software dependencies. Use this command "conda env create -f environment.yml" to create the environment for Pathfinder.

A.3.4 Data sets. Run "download.sh" to download the traces we tested in our paper which was originally provided by ML-based Data Prefetching Competition [2].

A.4 Installation

1. Clone Pathfinder GitHub repo:
   $ git clone https://github.com/linjiaty/Pathfinder.git
2. Create a conda environment
   $ cd ChampSim
   $ conda env create -f environment.yml
   $ conda activate snn-champ_test
3. Build Pathfinder
   $ ./ml_prefetch_sim.py build

A.5 Experiment workflow

Generate prefetch files using Pathfinder. The prefetch files will be generated in the 'pathfinder_prefetches_gap_spec' folder and the results files will be in the 'results' folder.

$ run_pathfinder_gap_spec.sh

A.6 Evaluation and expected results

- **Evaluation:** We evaluate Pathfinder on three metrics: IPC, Accuracy, and Coverage. All the IPC numbers from Pathfinder, Best Offset Prefetcher, SISB Prefetcher, and No Prefetch will be generated in the results folder. Accuracy and Coverage can be calculated by the formula below:
  $Accuracy = \frac{LLC\ PREFETCH\ ISSUED}{LLC\ PREFETCH\ REQUEST}$
  $Coverage = \frac{LLC\ PREFETCH\ USEFUL}{LLC\ LOAD\ ACCESS}$
- **Expected Results:** The results files will contain the results generated by Pathfinder prefetches, Best Offset Prefetches (BO), SISB Prefetches (SISB), and No Prefetch. As seen in Figure 4, Pathfinder exhibits better IPC than BO and is competitive with SISB.

A.7 Additional Baseline Prefetchers

We have two more baseline prefetchers that have their standalone GitHub repo.

- **Voyager** can be tested in the below GitHub repo, which is also implemented in the same ML-based ChampSim simulator as ours. (Voyager's model needs a long time to train on GPU. Start the training process early.) After generating prefetch files from Voyager, you can create a new folder in Pathfinder, put the prefetch files into it, comment out the command to generate Pathfinder prefetches, replace the corresponding directory/file names in run_pathfinder_gap_spec.sh, then run it to generate result files.
  GitHub Link: https://github.com/Quangmire/voyager
- **Pythia** is ported into LLC in an ML-based ChampSim simulator with thorough tests to serve as an LLC prefetcher for a fair comparison among all prefetchers. Run "trace_script.sh" to generate result files.
  GitHub Link: https://github.com/linjiaty/Pythia_test.git
  Unmodified Pythia: https://github.com/CMUSAFARI/Pythia
- **SPP** The results of SPP prefetcher will be available in the results file of Pythia_test.

A.8 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

References

[1] The 2nd cache replacement championship. https://crc2.ece.tamu.edu/.

[2] Ml-based data prefetching competition. https://sites.google.com/view/mlarchsys/isca-2021/ml-prefetching-competition.

[3] J.L. Baer and T.F. Chen. An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty. In *Proceedings of Supercomputing*, 1991.

[4] Mohammad Bakhshalipour, Mehran Shakerinava, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. Bingo spatial data prefetcher. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 399–411. IEEE, 2019.

[5] R. Balasubramonian, A.B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM TACO*, 14(2), 2017.

[6] Rahul Bera, Konstantinos Kanellopoulos, Anant V. Nori, Taha Shahroodi, Sreenivas Subramoney, and Onur Mutlu. Pythia: A customizable hardware prefetching framework using online reinforcement learning. *CoRR*, abs/2109.12021, 2021.

[7] Guo-qiang Bi and Mu-ming Poo. Synaptic modifications in cultured hippocampal neurons: dependence on spike timing, synaptic strength, and postsynaptic cell type. *Journal of neuroscience*, 18(24):10464–10472, 1998.

[8] Andrew S. Cassidy, Jun Sawada, Paul Merolla, John V. Arthur, Rodrigo Alvarez-Icaza, Filipp Akopyan, Bryan L. Jackson, and Dharmendra S. Modha. Truenorth: A high-performance, low-power neurosynaptic processor for multi-sensory perception, action, and cognition. 2016.

[9] Chi F Chen, S-H Yang, Babak Falsafi, and Andreas Moshovos. Accurate and complexity-effective spatial pattern prediction. In *10th International Symposium on High Performance Computer Architecture (HPCA'04)*, pages 276–287. IEEE, 2004.

[10] Tien-Fu Chen and Jean-Loup Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE transactions on computers*, 44(5):609–623, 1995.

[11] Mike Davies, Narayan Srinivasa, Tsung-Han Lin, Gautham Chinya, Prasad Joshi, Andrew Lines, Andreas Wild, Hong Wang, and Deepak Mathaikutty. Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro*, PP:1–1, 01 2018.

[12] P. Diehl and M. Cook. Unsupervised learning of digit recognition using spike-timing-dependent plasticity. In *Frontiers in Computational Neuroscience*, 2015.

[13] Peter Diehl and Matthew Cook. Unsupervised learning of digit recognition using spike-timing-dependent plasticity. *Frontiers in Computational Neuroscience*, 9, 2015.

[14] John WC Fu, Janak H Patel, and Bob L Janssens. Stride directed prefetching in scalar processors. *ACM SIGMICRO Newsletter*, 23(1-2):102–110, 1992.

[15] Jeff Gehlhaar. Neuromorphic processing: a new frontier in scaling computer architecture. In *ASPLOS*, pages 317–318, 2014.

[16] Milad Hashemi, Kevin Swersky, Jamie Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. Learning memory access patterns. In *International Conference on Machine Learning*. PMLR, 2018.

[17] Milad Hashemi, Kevin Swersky, Jamie A. Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. Learning memory access patterns. *CoRR*, abs/1803.02329, 2018.

[18] Hananel Hazan, Daniel J. Saunders, Hassaan Khan, Devdhar Patel, Darpan T. Sanghavi, Hava T. Siegelmann, and Robert Kozma. Bindsnet: A machine learning-oriented spiking neural networks library in python. *Frontiers in Neuroinformatics*, 12, 2018.

[19] Sorin Iacobovici, Lawrence Spracklen, Sudarshan Kadambi, Yuan Chou, and Santosh G Abraham. Effective stream-based and execution-based data prefetching. In *Proceedings of the 18th annual international conference on Supercomputing*, pages 1–11, 2004.

[20] Akanksha Jain and Calvin Lin. Linearizing irregular memory accesses for improved correlated prefetching. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 247–259, 2013.

[21] Akanksha Jain and Calvin Lin. Linearizing irregular memory accesses for improved correlated prefetching. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 247–259, 2013.

[22] Teresa L Johnson, Matthew C Merten, and Wen-Mei W Hwu. Run-time spatial locality detection and optimization. In *Proceedings of 30th Annual International Symposium on Microarchitecture*, pages 57–64. IEEE, 1997.

[23] Norman P Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *ACM SIGARCH Computer Architecture News*, 18(2SI):364–373, 1990.

[24] M.M. Khan, D.R. Lester, L.A. Plana, A. Rast, X. Jin, E. Painkras, and S.B. Furber. Spinnaker: Mapping neural networks onto a massively-parallel chip multiprocessor. In *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, pages 2849–2856, 2008.

[25] Saeed Reza Kheradpisheh, Mohammad Ganjtabesh, Simon J. Thorpe, and Timothée Masquelier. Stdp-based spiking deep neural networks for object recognition. *CoRR*, abs/1611.01421, 2016.

[26] Jinchun Kim, Seth H Pugsley, Paul V Gratz, AL Narasimha Reddy, Chris Wilkerson, and Zeshan Chishti. Path confidence based lookahead prefetching. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE.

[27] An-Chow Lai, Cem Fide, and Babak Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *Proceedings 28th annual international symposium on computer architecture*, pages 144–154. IEEE, 2001.

[28] Beiye Liu, Yiran Chen, Bryant Wysocki, and Tingwen Huang. Reconfigurable neuromorphic computing system with memristor-based synapse design. *Neural Processing Letters*, 41(2):159–167, 2015.

[29] Chenchen Liu, Bonan Yan, Chaofei Yang, Linghao Song, Zheng Li, Beiye Liu, Yiran Chen, Hai Li, Qing Wu, and Hao Jiang. A spiking neuromorphic design with resistive crossbar. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2015.

[30] Pierre Michaud. A best-offset prefetcher. In *2nd Data Prefetching Championship*, 2015.

[31] Pierre Michaud. Best-offset hardware prefetching. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 469–480, 2016.

[32] Sparsh Mittal. A survey of recent prefetching techniques for processor caches. *ACM Computing Surveys (CSUR)*, 49(2):1–35, 2016.

[33] Surya Narayanan, Ali Shafiee, and Rajeev Balasubramonian. Inxs: Bridging the throughput and energy gap for spiking neural networks. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 2451–2459, 2017.

[34] Surya Narayanan, Karl Taht, Rajeev Balasubramonian, Edouard Giacomin, and Pierre-Emmanuel Gaillardon. Spinalflow: An architecture and dataflow tailored for spiking neural networks. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 349–362, 2020.

[35] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[36] Leeor Peled, Shie Mannor, Uri Weiser, and Yoav Etsion. Semantic locality and context-based prefetching using reinforcement learning. *SIGARCH Comput. Archit. News*, 43(3S):285–297, jun 2015.

[37] Seth H Pugsley, Zeshan Chishti, Chris Wilkerson, Peng-fei Chuang, Robert L Scott, Aamer Jaleel, Shih-Lien Lu, Kingsum Chow, and Rajeev Balasubramonian. Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 626–637, 2014.

[38] Seth H Pugsley, Zeshan Chishti, Chris Wilkerson, Peng-fei Chuang, Robert L Scott, Aamer Jaleel, Shih-Lien Lu, Kingsum Chow, and Rajeev Balasubramonian. Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 626–637. IEEE, 2014.

[39] A. Seznec. A New Case for the TAGE Branch Predictor. In *Proceedings of MICRO*, 2011.

[40] M. Shevgoor, S. Koladiya, R. Balasubramonian, S. Pugsley, C. Wilkerson, and Z. Chishti. Efficiently Prefetching Complex Address Patterns. In *Proceedings of MICRO*, 2015.

[41] Zhan Shi, Akanksha Jain, Kevin Swersky, Milad Hashemi, Parthasarathy Ranganathan, and Calvin Lin. A hierarchical neural model of data prefetching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 861–873, New York, NY, USA, 2021. Association for Computing Machinery.

[42] Zhan Shi, Akanksha Jain, Kevin Swersky, Milad Hashemi, Parthasarathy Ranganathan, and Calvin Lin. Voyager github repository. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 861–873, New York, NY, USA, 2021. Association for Computing Machinery.

[43] Ivan Sklenář. Prefetch unit for vector operations on scalar computers. *ACM SIGARCH Computer Architecture News*, 20(4):31–37, 1992.

[44] Stephen Somogyi, Thomas F Wenisch, Anastasia Ailamaki, and Babak Falsafi. Spatio-temporal memory streaming. *ACM SIGARCH Computer Architecture News*, 37(3):69–80, 2009.

[45] Stephen Somogyi, Thomas F Wenisch, Anastassia Ailamaki, Babak Falsafi, and Andreas Moshovos. Spatial memory streaming. *ACM SIGARCH Computer Architecture News*, 34(2):252–263, 2006.

[46] Ajitesh Srivastava, Aggelos Lazaris, Benjamin Brooks, Rajgopal Kannan, and Viktor Prasanna. Predicting memory accesses: the road to compact ml-driven prefetcher. pages 461–470, 09 2019.

[47] Synopsys. Synopsys Design Compiler User Guide. http://www.synopsys.com/\Tools/Implementation/RTLSynthesis/DCUltra/Pages/.

[48] Tianqi Tang, Lixue Xia, Boxun Li, Rong Luo, Yiran Chen, Yu Wang, and Huazhong Yang. Spiking neural network with rram: Can we use it for real-world application? In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 860–865, 2015.

[49] WikiChip. Ryzen 7 2700X - AMD. https://en.wikichip.org/wiki/amd/ryzen_7/2700x.

[50] Michael Wu, Ketaki Joshi, Andrew Sheinberg, Guilherme Cox, Anurag Khandelwal, Raghavendra Pradyumna Pothukuchi, and Abhishek Bhattacharjee. Prefetching using principles of hippocampal-neocortical interaction. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, HOTOS '23, page 53–60, New York, NY, USA, 2023. Association for Computing Machinery.

[51] Pengmiao Zhang, Rajgopal Kannan, Ajitesh Srivastava, Anant V Nori, and Viktor K Prasanna. Resemble: reinforced ensemble framework for data prefetching. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14. IEEE, 2022.

[52] Pengmiao Zhang, Ajitesh Srivastava, Benjamin Brooks, Rajgopal Kannan, and Viktor K. Prasanna. Raop: Recurrent neural network augmented offset prefetcher. In *The International Symposium on Memory Systems*, MEMSYS 2020, page 352–362, New York, NY, USA, 2020. Association for Computing Machinery.