A Hierarchical Neural Model of Data Prefetching

Zhan Shi
zshi17@cs.utexas.edu
The University of Texas at Austin
USA

Akanksha Jain
akanksha@cs.utexas.edu
The University of Texas at Austin
USA

Kevin Swersky
kswersky@google.com
Google Research
USA

Milad Hashemi
miladh@google.com
Google Research
USA

Parthasarathy Ranganathan
parthas@google.com
Google Research
USA

Calvin Lin
lin@cs.utexas.edu
The University of Texas at Austin
USA

ABSTRACT

This paper presents Voyager, a novel neural network for data prefetching. Unlike previous neural models for prefetching, which are limited to learning delta correlations, our model can also learn address correlations, which are important for prefetching irregular sequences of memory accesses. The key to our solution is its hierarchical structure that separates addresses into pages and offsets and that introduces a mechanism for learning important relations among pages and offsets.

Voyager provides significant prediction benefits over current data prefetchers. For a set of irregular programs from the SPEC 2006 and GAP benchmark suites, Voyager sees an average IPC improvement of 41.6% over a system with no prefetcher, compared with 21.7% and 28.2%, respectively, for idealized Domino and ISB prefetchers. We also find that for two commercial workloads for which current data prefetchers see very little benefit, Voyager dramatically improves both accuracy and coverage.

At present, slow training and prediction preclude neural models from being practically used in hardware, but Voyager's overheads are significantly lower—in every dimension—than those of previous neural models. For example, computation cost is reduced by 15-20×, and storage overhead is reduced by 110-200×. Thus, Voyager represents a significant step towards a practical neural prefetcher.

CCS CONCEPTS

• **Computer systems organization → Architectures**.

KEYWORDS

Prefetching, Neural Networks, Attention Mechanism

ACM Reference Format:
Zhan Shi, Akanksha Jain, Kevin Swersky, Milad Hashemi, Parthasarathy Ranganathan, and Calvin Lin. 2021. 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 '21), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3445814.3446752

1 INTRODUCTION

Machine learning has provided insights into various hardware prediction problems, including branch prediction [22, 23, 49, 58] and cache replacement [24, 43, 50]. So it is natural to ask if machine learning (ML) could play a role in advancing the state-of-the-art in data prefetching, where improvements have recently been difficult to achieve. Unfortunately, data prefetching presents two challenges to machine learning that branch prediction and cache replacement do not.

First, data prefetching suffers from the *class explosion* problem. While branch predictors predict a binary output—Taken or Not Taken—and while cache replacement can be framed as a binary prediction problem [20, 24, 28, 43, 55]—a line has either high or low priority—prefetchers that learn delta correlations or address correlations have enormous input and output spaces. For example, for address correlation, also known as temporal prefetching, the inputs and outputs are individual memory addresses, so for a 64-bit address space, the model needs to predict from among tens of millions of unique address values. Such predictions cannot be handled by existing machine learning models for image classification or speech recognition, which traditionally have input and output spaces that are orders of magnitude smaller.[1]

Second, data prefetching has a labeling problem. Whereas branch predictors can be trained by the ground truth answers as revealed by a program's execution, and whereas cache replacement policies can be trained by learning from Belady's provably optimal MIN policy [20], data prefetchers have no known ground truth labels from which to learn. Thus, given a memory access m, the prefetcher could learn to prefetch any of the addresses that follow m. In machine learning parlance, it's not clear which *label* to use to train the ML model.

Previous work has made inroads into these problems. Most notably, Hashemi, et al [13] show that prefetching can be phrased as a classification problem and that neural models, such as LSTMs [14],[2] can be used for prefetching. But Hashemi, et al. focus on learning

[1]The large number of outputs cannot be handled by Perceptrons either. Perceptrons are by default designed for binary classification, so a typical method of dealing with multiple prediction outputs is to have n models, each of them separating one output class from the rest, where n is the number of outputs. Thus, for large output spaces, Perceptrons are both expensive and ineffective.

[2]LSTM stands for Long Short-Term Memory, and LSTMs are neural networks that can learn sequences of events.

delta correlations, or strides, among memory references, and their solution does not generalize to address correlation.

Since off-the-shelf neural models suffer from the two aforementioned problems, our goal in this paper is to develop a novel neural model that can learn both delta and address correlations. We tackle the two problems by developing a novel hierarchical neural network structure that exploits unique properties of data prefetching (described momentarily) that are not found in ML tasks in computer vision or natural language processing.

To solve the class explosion problem, we decompose address prediction into two sub-problems, namely, page prediction and offset prediction. Thus, while a program can have tens of millions of unique addresses, the total number of unique pages is only in the tens or hundreds of thousands, and the number of unique offsets is fixed at 64.

While this decomposition might appear to be both obvious and trivial, the naive decomposition leads to the *offset aliasing* problem in which all addresses with the same offset but different pages will alias with one another. More precisely, different addresses with the same offset will share the same offset embedding,[3] which limits the ability of neural networks to learn, because different data addresses pull the shared offset embedding towards different answers, resulting in poor performance. To address this issue, we use a novel attention-based *embedding layer* that allows the page prediction to provide context for the offset prediction. This context enables our shared embedding layer to differentiate data addresses without needing to learn a unique representation for every data address.

To solve the labeling problem, we observe that the problem is similar to the notion of localization for prefetchers. For example, temporal prefetchers, such as STMS [54] and ISB [19], capture the correlation between a pair of addresses, A and B, where A is the trigger, or input feature, and B is the prediction, or output label. The labeling problem is to find the most correlated output label B for the input feature A. In STMS, A and B are consecutive memory addresses in the global access stream. In ISB, which uses PC localization [35], A and B are consecutive memory addresses accessed by a common PC. But PC-localization is not always sufficient, for example, in the presence of data-dependent correlations across multiple PCs. Thus, to explore new forms of localization, we build into our neural model a *multi-label* training scheme that enables the model to learn from multiple possible labels. The key idea is that instead of providing a single ground truth label, the model can learn the label that is most predictable.

While this paper does not yet make neural models practical for use in hardware data prefetchers, it shows that it is possible to advance the state-of-the-art in data prefetching by using superior prefetching algorithms. In particular, this paper makes the following contributions:

- We advance the state-of-the-art in neural network-based prefetching by presenting Voyager,[4] a neural network model that can perform temporal prefetching. Our model uses a novel attention-based embedding layer to solve key challenges that arise from handling the large input and output spaces of correlation-based prefetchers.
- We outline the design space of temporal prefetchers by using the notion of features and localization, and we show that neural networks are capable of exploiting rich features, such as the history of data addresses.
- We are the first to demonstrate that LSTM-based prefetchers can outperform existing hardware prefetchers (see Section 2). Using a set of irregular benchmarks, Voyager achieves accuracy/coverage of 79.6%, compared with 57.9% for ISB [19], and Voyager improves IPC over a system with no prefetcher by 41.6%, compared with 28.2% for ISB. More significantly, on Google's *search* and *ads*, two applications that have proven remarkably resilient against hardware data prefetchers, our model achieves 37.8% and 57.5% accuracy/coverage, respectively, where an idealized version of ISB sees accuracy/coverage of just 13.8% and 26.2%, respectively.
Thus, our solution shows that significant headroom still exists for data prefetching, which is instructive for a problem for which there is no optimal solution.
- We also show that Voyager significantly outperforms previous neural prefetchers [13], producing accuracy/coverage of 79.6%, compared with Delta-LSTM's 56.8%, while also significantly reducing overhead in training cost, prediction latency, and storage overhead. For example, Voyager reduces training and prediction cost by 15-20×, and it reduces model size by 110-200×. Voyager's model size is smaller than those of non-neural state-of-the-art temporal prefetchers [3, 54, 57].

The remainder of this paper is organized as follows. Section 2 contrasts our work with both traditional data prefetchers and machine learning-based prefetchers. Section 3 then presents our probabilistic formulation of data prefetching, which sets the stage for the description of our our new neural prefetcher in Section 4. We then present our empirical evaluation in Section 5, before providing concluding remarks.

2 RELATED WORK

We now discuss previous work in data prefetching, which can be described as either *rule-based* or *machine learning-based*. The vast majority of prefetchers are *rule-based*, which means that they predict future memory accesses based on pre-determined learning rules.

2.1 Rule-Based Data Prefetchers

Many data prefetchers use rules that target sequential [16, 27, 44] or strided [2, 11, 17, 37, 41] memory accesses. For example, stream buffers [2, 27, 37] confirm a constant stride if some fixed number of consecutive memory accesses are the same stride apart. More recent prefetchers [32, 40] improve upon these ideas by testing a few pre-determined strides to select an offset that provides the best coverage. Offset-based prefetchers are simple and powerful, but their coverage is limited because they apply a single offset to all accesses. By contrast, Voyager can employ different offsets for different memory references.

[3]An embedding is an internal representation of input features within a neural network, and an embedding layer learns this representation during training such that features that behave similarly have similar embeddings.
[4]We name our system after the Voyager space probes, which were launched to extend the horizon of space exploration with no guarantees of what they would find.

Instead of predicting constant offsets, another class of prefetchers uses *delta correlation* to predict recurring delta patterns [36, 42]. For example, Nesbit et al.'s PC/DC prefetcher [36] and Shevgoor et al.'s Variable Delta Length Prefetcher predict these patterns by tracking deltas between consecutive accesses by the same PC. The Signature Path Prefetcher uses compressed signatures that encapsulate past addresses and strides within a page to generate the next stride [29]. Voyager is better equipped to predict these delta patterns, as it can take longer context—potentially spanning multiple spatial regions—to make a stride prediction.

Irregular prefetchers move beyond sequential and strided access patterns. Some irregular accesses can be captured by predicting recurring spatial patterns across different regions in memory [4, 7, 8, 25, 31, 47]. For example, the SMS prefetcher [47] learns recurring spatial footprints within page-sized regions and applies old spatial patterns to new unseen regions, and the Bingo prefetcher [4] uses longer address contexts to predict footprints.

Temporal prefetchers learn irregular memory accesses by memorizing pairs of correlated addresses, a scheme that is commonly referred to as *address correlation* [9, 53]. Early temporal prefetchers keep track of pairwise correlation of consecutive memory addresses in the global access stream [10, 15, 26, 35, 45, 54], but these prefetchers suffer from poor coverage and accuracy due to the poor predictability of the global access stream. More recent temporal prefetchers look for pairwise correlations of consecutive addresses in a PC-localized stream [19, 56, 57], which improves coverage and accuracy due to the superior predictability of the PC-localized stream. Instead of using PC-localization, Bakhshalipour, et al, improve the predictability in the global stream by extending pairwise correlation with one more address as features [3]. In particular, their Domino prefetcher predicts the next address by memorizing its correlation to the two past addresses in the global stream. All of these temporal prefetchers use a fixed localization scheme and a fixed method of correlating addresses. By contrast, Voyager leverages richer features and localizers in a data-driven fashion.

2.2 Machine Learning-Based Prefetchers

Peled et al., use a table-based reinforcement learning (RL) framework to explore the correlation between richer program contexts and memory addresses [38]. While the RL formulation is conceptually powerful, the use of tables is insufficient for RL because tables are sample inefficient and sensitive to noise in contexts. To improve the predictor, Peled et al. use a fully-connected feed-forward network [39] instead, and they formulate prefetching as a regression problem to train their neural network. Unfortunately, regression-based models are trained to arrive *close* to the ground truth label, but since a small error in a cache line address will prefetch the wrong line, being close is not useful for prefetching.

Hashemi et al. [13] were the first to formulate prefetching as a classification problem and to use LSTMs for building a prefetcher. However, to reduce the size of the output space, their solution can only learn deltas within a spatial region, so their LSTM cannot perform irregular data prefetching. Moreover, their paper targets a machine learning audience, so they use a machine learning evaluation methodology: Training is performed offline rather than in a microarchitectural simulator, and their metrics do not include IPC and do not translate to a practical setting. For example, a prefetch is considered correct if any one of the ten predictions by the model match the next address, thus ignoring practical considerations of accuracy and timeliness. Recent work improves the efficiency of this delta-based LSTM at the cost of lower coverage [48].

Our work differs from prior work in several ways. First, our work is the first to show the IPC benefits of using an LSTM prefetcher. Second, our work is the first neural model that combines both delta patterns and address correlation. Third, our multi-labeling scheme can provide a richer set of labels, while allowing the model to pick the label that it finds the most predictable. Finally, our model is significantly more compact and significantly less computationally expensive than prior neural solutions.

3 PROBLEM FORMULATION

To lay a strong foundation for our ML solution, we first formulate data prefetching as a probabilistic prediction problem and view its output as a *probability distribution*. This formulation will help us motivate the use of ML models, because machine learning, especially deep learning, provides a flexible framework for modeling probability distributions. It will also allow us to view a wide range of existing data prefetchers—including temporal prefetchers and stride prefetchers—within a unified framework.

3.1 Probabilistic Formulation of Temporal Prefetching

The goal of temporal prefetching is to exploit correlations between consecutive addresses to predict the next address. Therefore, temporal prefetching can be viewed as a classification problem where each address is a *class*, and the learning task is to learn the probability that an address $Addr$ will be accessed given a history of past events, such as the occurrence of memory accesses $Access_1, Access_2, ..., Access_t$ up to the current timestamp t:

$$P(Addr|Access_1, Access_2, ..., Access_t) \tag{1}$$

In ML terminology, the historical events ($Access_1$, $Access_2$ ..., $Access_t$) are known as *input features*, and the future event ($Addr$) is known as the model's *output label*.

All previous temporal prefetchers [3, 19, 46, 53, 54, 56, 57] can be viewed as instances of this formulation with different input features and output labels. For example, STMS [54] learns the temporal correlation between consecutive addresses in the global memory access stream, so its output label is the next address in the global memory access stream. Thus, STMS tries to learn the following probability distribution:

$$P(Addr_{t+1}|Addr_t) \tag{2}$$

ISB [19] implements PC localization, which improves upon STMS by providing a different output label, namely, the next address by the same program counter (PC). Thus, ISB tries to learn the following probability distribution:

$$P(Addr_{PC}|Addr_t) \tag{3}$$

where $Addr_{PC}$ is the next address that will be accessed by the PC that just accessed $Addr_t$.

Domino [3] instead improves upon STMS by using a different input feature, using the previous two addresses to predict the next address in the global memory access stream:

$$P(Addr_{t+1} | Addr_{t-1}, Addr_t) \tag{4}$$

3.2 Probabilistic Formulation of Stride Prefetching

Stride prefetchers can also be described under this probabilistic framework by incorporating strides or deltas in our formulation. For example, a stride prefetcher detects the constant stride pattern by observing the strides at consecutive timestamps t and $t + 1$:

$$P(Stride_{t+1} | Stride_t) \tag{5}$$

As with ISB, the idea of using a per-PC output (PC localization) is also used by the IP stride prefetcher.

$$P(Stride_{PC} | Stride_t) \tag{6}$$

The VLDP prefetcher [42] looks at a history of past deltas and selects the most likely deltas.

$$P(Stride_{t+1} | Stride_{t_0}, Stride_{t_1}, ..., Stride_{t_n}) \tag{7}$$

Hashemi et al.'s neural prefetcher [13] adopts a similar formulation. Given a history length l, it learns the following distribution:

$$P(Stride_{t+1} | Stride_{t-l}, Stride_{t-l+1}, ..., Stride_t) \tag{8}$$

In general, our probabilistic formulation of prefetching defines the input features (the historical event) and the output label (the future event). Unlike previous learning-based work that focuses on limited features and focuses on the prediction of the global stream, we seek to improve the prediction accuracy by exploring the design choices of both the input features and the output labels.

4 OUR SOLUTION: VOYAGER

This section describes Voyager, our neural model for performing data prefetching. We start by presenting a high-level overview of the model. We then describe the three key innovations in Voyager's model design. First, to enable the model to learn temporal correlations among millions of addresses, Voyager uses a hierarchical neural structure, with one part of the model predicting page addresses and the other part predicting offsets within a page. This hierarchical structure is described in Section 4.2. Second, to cover compulsory misses, Voyager uses a vocabulary[5] that includes both addresses and deltas. This ability to use both addresses and deltas is described in Section 4.3. Third, Voyager adopts a multi-label training scheme, so that instead of predicting the next address in the global address stream, Voyager is trained to predict the most predictable address from multiple possible labels. This multi-label training scheme is described in Section 4.4.

[5]A neural network's vocabulary is the set of words that the model can admit as input and can produce as output.

4.1 Overall Design and Workflow

Figure 1 shows that Voyager takes as input a sequence of memory accesses and produces as output the next address to be prefetched. Each memory access in the input is represented by a PC and an address, and each address is split into a page address and an offset within the page.

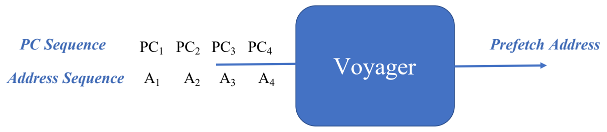


Figure 1: Overview of Voyager.

Figure 2 shows Voyager's neural architecture. Since the inputs (PCs, page addresses, offsets) have no numerical meaning, the first layer computes *embeddings* that translate each input into a real number such that inputs that behave similarly have similar embeddings. Our first embedding layer computes independent embeddings for PCs, pages, and offsets, and our second embedding layer (shown in purple) is the novel page-aware offset embedding layer that revises the offset's representation (or embedding) to be page-aware. (See Section 4.2 for details.) The next layer takes these embeddings as input and uses two separate *LSTMs* to predict the embeddings for candidate output pages and offsets, respectively. Finally, the candidates from the two LSTMs are fed into a linear layer with a softmax activation function,[6] producing a probability distribution over the predicted pages and offsets. The page and offset pair with the highest probability is chosen as the address to prefetch. Table 1 shows all the hyperparameters used in Voyager. To emulate a hardware prefetcher, the entire model is trained *online*, which means that it is trained continuously as the program runs (see Section 5.1 for more details).

Table 1: Hyperparameters for training Voyager.

Sequence length (i.e. history length)	16
Learning rate	0.001
Learning rate decay ratio	2
Embedding size for PC	64
Embedding size of page	256
Embedding size of offset	25600
# Experts	100
Page and offset LSTM # layers	1
Page and offset LSTM # units	256
Dropout keep ratio	0.8
Batch size	256
Optimizer	Adam

4.2 Hierarchical Neural Structure

Before explaining our novel page-aware offset embedding layer, this section first motivates the need for a hierarchical neural model.

[6]The softmax function converts a vector of real numbers into an equal-sized vector whose values sum to 1. Thus, the softmax function produces values that can be interpreted as probabilities.

4.2.1 *Motivation.*

Table 2 shows that the number of unique addresses in our benchmark programs ranges from hundreds of thousands to tens of millions. These numbers greatly surpass the number of unique categories in traditional ML tasks, such as natural language processing, where the typical vocabulary size is 100K. Large vocabularies are problematic for two reasons: (1) The explosion of memory addresses leads to an increase in memory usage that precludes the training of neural networks [13, 43], and (2) the large number of unique memory addresses makes it difficult to train the model because each address appears just a few times. By contrast, Table 2 shows that the number of pages is in the tens of thousands and is therefore much more manageable.

Table 2: Benchmark statistics.

Benchmark	# PCs	# Addresses	# Pages
astar	192	0.15M	29.9K
bfs	828	0.16M	4.1K
cc	529	0.26M	4.3K
mcf	169	4.58M	91.1K
omnetpp	1101	0.48M	36.3K
pr	650	0.27M	4.2K
soplex	2129	0.36M	12.3K
sphinx	1519	0.13M	4.3K
xalancbmk	2071	0.34M	25.3K
search	6729	0.91M	22.4K
ads	21159	1.4M	28.7K

A naive model would treat page prediction and offset prediction as independent problems: At each step of the memory address sequence, the input would be represented as a concatenation of the page address and the offset address, each of which would be fed to two separate LSTMs—a page LSTM and an offset LSTM—to generate the page and offset of the future address.

Unfortunately, the naive splitting of addresses into pages and offsets leads to a problem that we refer to as *offset aliasing*. To understand this aliasing problem, consider two addresses X and Y that have different page numbers but the same offset O. With a naive splitting, the offset LSTM will see the same input O for both X and Y and will be unable to distinguish the offest of X from the offest of Y, leading to incorrect predictions. Because there are only 64 possible offsets, the offset aliasing problem is quite common. Our novel *page-aware offset embedding layer* resolves this issue by providing every offset with context about the page of the input address.

4.2.2 *Page-Aware Offset Embedding.*

The ideal offset embedding not only represents the offset but also includes some context about the page that it resides on. The analogy in natural language is polysemy where multiple meanings exist for a word, and the actual meaning depends on the context in which the word is used; without this context, the models learn an average behavior of multiple distinct meanings, which is not useful. To make the offset (word) aware of the page (context), we take inspiration from the machine learning notion of *mixtures of experts* [18]. Intuitively, a word with multiple meanings can be handled by multiple experts, with each expert corresponding to one meaning. Depending on the context in which the word is used, the appropriate expert will be chosen to represent the specific meaning. Thus, our *page-aware offset embedding mechanism* uses a mixture of experts, where each expert for an offset represents a specific page-aware characteristic of that offset. In the worst case, the number of experts would equal to the number of pages, but in reality, the number of experts only needs to be large enough to capture the important behaviors. We empirically find that this number varies from 5 to 100 across benchmarks.

Figure 3 illustrates the page-aware offset embedding mechanism in more detail. The core mechanism is an *attention layer* [52] that takes as input a query, a set of keys and a set of values, and it measures the correlation between the query and the keys. The output of the attention layer is a weighted sum of the values, such that the weights are the learned correlations between the query and keys. In our case, the attention layer is optimized using a scoring function that takes the page as the *query*, and the offset embeddings for each expert as both the *keys* and *values*. Given a query (the page embedding), the layer computes the page's correlation with each key (the offset embedding) and produces a probability vector that represents this correlation. The final output offset embedding is a sum over the input page-agnostic embeddings, weighted by these correlation probabilities. This mechanism is known as soft attention and allows us to use backpropagation to learn the correlation vectors.

Formally, we can think of the offset embedding as one large vector, and we can think of each expert as being one partition of this vector (see Figure 3). When we set the ratio between page embedding size and total offset embedding size to be n, corresponding to n experts, the mechanism can be defined as

$$a_t(o,s) = \frac{exp(f \cdot score(h_p, h_{o,s}))}{\sum_{s'} exp(f \cdot score(h_p, h_{o,s'}))} \tag{9}$$

$$h'_o = \sum_s a_t(o,s) h_{o,s} \tag{10}$$

where f is a scaling factor that ranges from 0 to 1; h_p is the page embedding; $h_o = [h_{o,0}, h_{o,1}, ..., h_{o,n}]$ is the offset embedding, where $h_{o,i}$ is the embedding of the i^{th} expert; and h'_o is the page-aware offset embedding generated by the attention mechanism. Empirically, we set the size of the offset embedding $|h_o|$ to be 5-100× of that of the page embedding $|h_p|$. In the example in Figure 3, we use a dot-product attention layer with a 200-dimension (d) page embedding ($|h_p| = 200$) and 1000-dimension (d) offset embedding ($|h_o| = 1000$). The 1000-d offset embedding $|h_o|$ is divided into 5 expert embeddings ($n = 5$), each of which is the same size as the page embedding used to perform the attention operation. Attention weights $a_t(o,s)$ are computed as the dot product of the page embedding and each of the offset expert embeddings, and a final page-aware offset embedding h'_o is obtained by a weighted sum of all the offset expert embeddings $h_{o,k}, k = 0, 1, ..., n$.

Since the embedding layer is the primary storage and computation bottleneck for networks with a large number of classes, the page-aware offset embedding dramatically reduces Voyager's size and dramatically reduces the number of parameters to learn. This reduction thus simplifies the model and reduces training overhead. In Section 5 we show that Voyager improves model efficiency—in

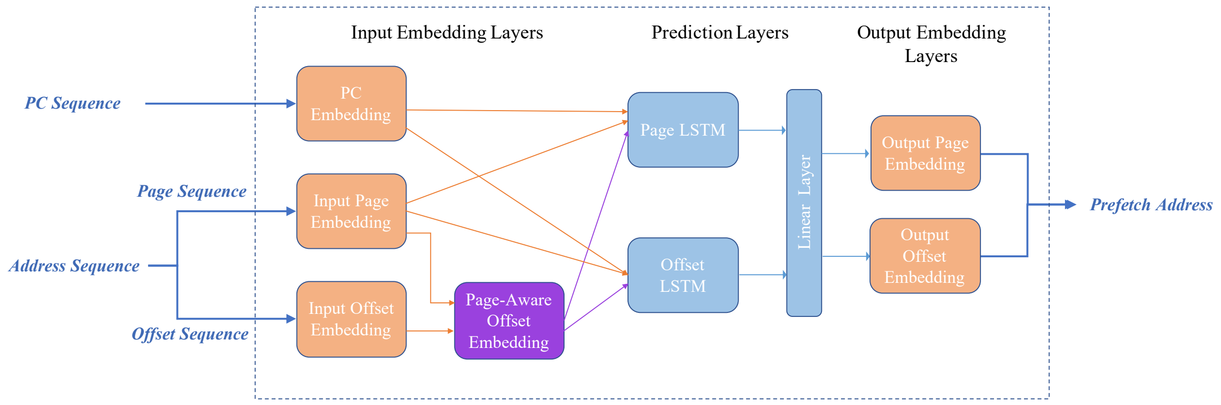


Figure 2: Voyager's Model Architecture.

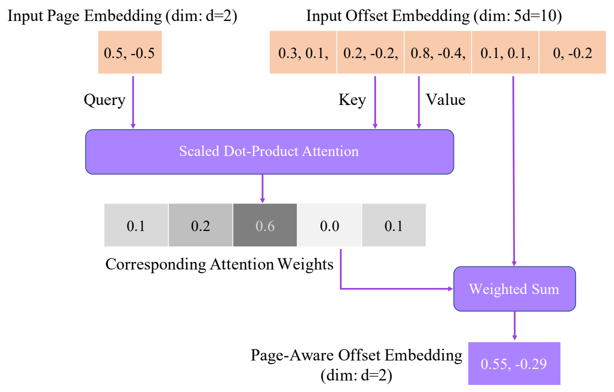

Input Page Embedding (dim: d=2)
Input Offset Embedding (dim: 5d=10)
0.5, -0.5
0.3, 0.1,
0.2, -0.2,
0.8, -0.4,
0.1, 0.1,
0, -0.2
Query
Key
Value
Scaled Dot-Product Attention
0.1
0.2
0.6
0.0
0.1
Corresponding Attention Weights
Weighted Sum
Page-Aware Offset Embedding (dim: d=2)
0.55, -0.29

Figure 3: Page-aware offset embedding with the dot-product attention mechanism. The vector values are calculated with a simplified dot-product attention without the scale factor or linear mappings. In this example, the 3^{rd} chunk of the offset embedding $(0.8, -0.4)$ correlates the most with the page embedding $(0.5, -0.5)$ and therefore contributes the most (normalized attention weight: 0.6) to the final page-aware offset embedding $(0.55, -0.29)$.

terms of computational cost and storage overhead—by an order of magnitude when compared to previous neural-based solutions [13].

4.3 Covering Compulsory Misses

We have so far explained how Voyager can learn address correlations, but address correlation-based prefetching has two limitations. First, it cannot handle compulsory misses, which are common in benchmarks with large memory footprints, such as *mcf* and *search*. Second, it is not worth learning correlations for addresses that occur infrequently. Since delta correlations *can* be used to prefetch compulsory misses, we address both issues by using deltas to represent correlations involving infrequently appearing addresses.

In particular, we enhance Voyager's vocabulary to include deltas. Addresses that have low frequency (they occur fewer than 2 times) are represented using the deltas of their page and offset from the previous page and offset respectively; infrequent addresses are identified by a profiling pass over the trace. To distinguish addresses from deltas, the delta page entries in the vocabulary are marked with a special symbol (e.g. the entry value starts with 'd'). For example, if X is an address that occurs infrequently, then we would represent the address sequence A, B, X as $A, B, d : X - B$, where X has been replaced by a delta value. In this way, our neural model will not attempt to learn address correlations for infrequent addresses, and it will be able to learn some delta correlations that can be used to prefetch some compulsory misses. Since we use deltas for infrequent addresses, our model needs just a small number of deltas. For example, we find that 10 deltas can cover 99% of the compulsory misses in *mcf*, whereas previous solutions [13] need millions of deltas.

4.4 Multi-Label Training Scheme

As explained in Section 1, data prefetchers do not have access to ground truth labels for training. We find that different labeling schemes work well for different workloads: Spatial labeling schemes work well for workloads that have spatial memory access patterns, and PC-based labeling schemes work well on pointer-based workloads. Some workloads have a mix of access patterns that require multiple labeling schemes.

We formulate this problem as *multi-label classification* [51]. Unlike traditional single-label multi-class classification, where each training sample is associated with a single output label, in multi-label classification, each training sample is associated with a set of labels. In our formulation, we provide each training sample in Voyager with the following candidate labels: (1) *global* represents the next address in the global stream, (2) *PC* represents the next address by the same PC, (3) *basic block* represents the next address by the PCs from the current basic block, (4) *spatial* represents the next address within a spatial offset of 256 [32], and (5) *co-occurrence* represents the address that occurs most often in the future window of 10 memory accesses. Figure 4 shows an example with multiple labeling schemes: When address A is seen, each of the candidates (B, C, E) are provided as a label for A.

To train with multiple labels, the main modification in the neural network is the design of the loss function. Instead of using the

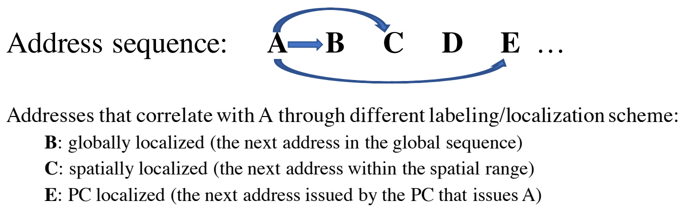

Address sequence:
A
B
C
D
E ...
Addresses that correlate with A through different labeling/localization scheme:
B: globally localized (the next address in the global sequence)
C: spatially localized (the next address within the spatial range)
E: PC localized (the next address issued by the PC that issues A)

Figure 4: An example of multiple labeling schemes. At address A, multiple future addresses are correlated with address A through different labeling schemes and they all are considered as potential outputs.

softmax loss function that normalizes the probability distribution over all potential outputs, the model is trained with the *binary cross entropy* (BCE) loss function [34]. BCE uses a sigmoid function to estimate the binary probability distribution of each individual candidate label, predicting whether or not it is likely to appear. Voyager's inference differs slightly from a typical multi-label classification task, as it selects the candidate with the highest probability instead of all candidates that pass a pre-determined threshold [51]. Thus, Voyager leverages the benefits of different labeling schemes and selects the most predictable label to make its prediction.

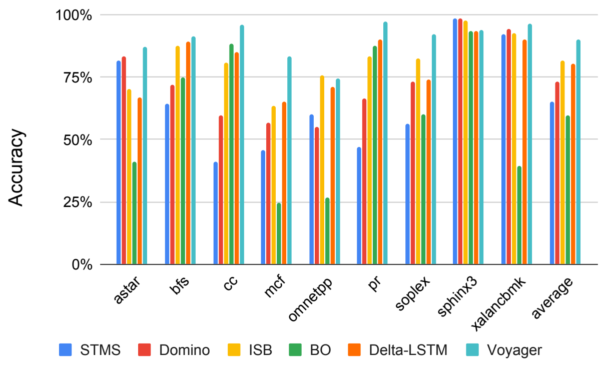


Figure 5: Accuracy.

5 EVALUATION

This section evaluates our ideas by comparing Voyager against both practical prefetchers and neural prefetchers.

5.1 Methodology

Neural networks are typically trained *offline* on a corpus of inputs, but since we want to evaluate Voyager as a hardware prefetcher, we train it *online* as the program executes. In particular, Voyager (and the baseline machine learning-based prefetchers) is trained for an epoch of 50 million instructions, and it uses this trained model to make predictions for the next epoch of 50 million instructions. Thus, the model is constantly being trained in one epoch for use in the next epoch. No inference is performed in the first epoch. This evaluation methodology contrasts sharply with that used by prior evaluations of machine learning-based prefetchers, where

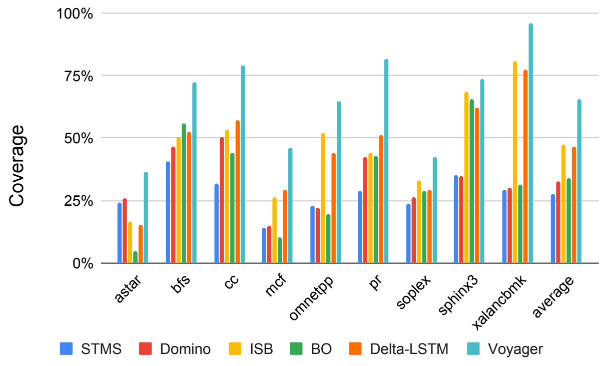


Figure 6: Coverage.

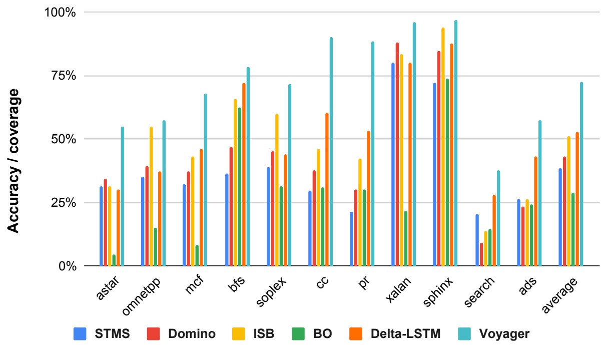


Figure 7: Unified accuracy/coverage, including Google's *search* and *ads*.

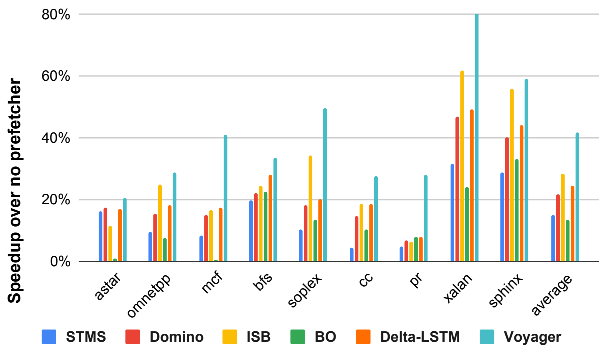


Figure 8: IPC.

the models are trained offline on one portion of the benchmark's execution and tested on a different portion of the benchmark's execution.

Simulator. We evaluate our models using the simulation framework released by the 2nd JILP Cache Replacement Championship (CRC2), which is based on ChampSim [21]. ChampSim models a

4-wide out-of-order processor with an 8-stage pipeline, a 128-entry reorder buffer and a three-level cache hierarchy. Table 3 shows the parameters for our simulated memory hierarchy.

Table 3: Simulation configuration.

L1 I-Cache	64 KB, 4-way, 3-cycle latency
L1 D-Cache	64 KB, 4-way, 3-cycle latency
L2 Cache	512 KB, 8-way, 11-cycle latency
LLC per core	2MB, 16-way, 20-cycle latency
DRAM	tRP=tRCD=tCAS=20 2 channels, 8 ranks, 8 banks 32K rows, 8GB/s bandwidth per core

All prefetchers are situated at the last-level cache (LLC), which means that their inputs are LLC accesses, and the prefetched entries are also inserted in the LLC.

Benchmarks. We evaluate Voyager and the baselines on a set of irregular benchmarks from the SPEC06 and GAP benchmark suites [5]. In particular, we use irregular benchmarks on which an oracle prefetcher that always correctly prefetches the next load produces at least a 10% IPC improvement over a baseline with no prefetching. This is the same methodology as used by previous work [19, 56, 57]. For each benchmark, we use SimPoint [12] to generate traces of length 250 million instructions. We use the reference input set for SPEC06 and input graphs of size 2^{17} nodes for GAP.

To evaluate Voyager on more challenging workloads, we also use Google's *search* and *ads*, two state-of-the-art enterprise-scale applications. Our *search* and *ads* results come from memory traces of production Google servers [6]; the traces use virtual addresses and only include memory instructions. With just memory instructions, the traces are not suitable for ChampSim, so we cannot simulate IPC numbers, just accuracy and coverage.

Baseline Prefetchers. We compare Voyager against spatial prefetchers (the Best Offset Prefetcher (BO) [32]), temporal prefetchers (STMS [54], ISB [19] and Domino [3]), and impractical neural prefetchers (Delta-LSTM [13]). Since our goal is to evaluate the prediction capabilities of different solutions, *we use idealized implementations of all baselines, so there are no constraints on model storage or off-chip metadata, and all storage is accessed with no cost.* Our baselines are particularly optimistic for the temporal prefetchers, which typically require 10-100M of off-chip metadata, so practical implementations would incur the latency and traffic overhead of accessing this off-chip metadata.

Metrics. We evaluate our solutions by comparing their accuracy, coverage, and IPC over a system with no prefetcher. For a fair evaluation, our IPC numbers do not consider the latency or storage cost of generating a prefetch address for any of the evaluated prefetchers. However, all prefetch requests are simulated accurately, and the IPC numbers accurately capture the impact of prefetcher accuracy and timeliness. We also include a comparison at higher degrees to evaluate the impact of aggressive prefetching.

Unfortunately, it's difficult to simulate Google's *search* and *ads* in a microarchitectural simulator, so we cannot directly compute coverage, accuracy, and IPC for these workloads. Therefore, to evaluate Voyager's effectiveness outside a microarchitectural simulator, we follow Srivastava et al. [48] and present additional data using a new unified definition of accuracy/coverage, in which the model's prediction is considered to be correct *only when it correctly predicts the next load address.* This metric unifies accuracy and coverage because each correct prediction improves both accuracy (as it is correct) and coverage (as the next address is covered). The value of this metric can also be interpreted as the percentage of addresses that are predicted to be prefetched.

This combined metric is also important for training Voyager, because neural models need to be trained using a single objective function—as opposed to having separate objective functions for coverage and accuracy. From a prediction perspective, this unified metric means that Voyager is designed to improve both accuracy and coverage simultaneously.

We also compare the overhead of Voyager, including computational cost and model size, against both a non-hierarchical neural network implementation [13] and a temporal prefetcher [19].

5.2 Comparison with Prior Art

Figures 5 and 6 show that Voyager improves the accuracy of our SPEC and GAP benchmarks from 81.6% to 90.2% and coverage from 47.2% to 65.7%.

Figure 7 compares the unified accuracy/coverage metric on all benchmarks, including Google's *search* and *ads* benchmarks. We see that Voyager is particularly effective for Google's *search* and *ads* where it improves accuracy/coverage to 37.8% and 57.5%, respectively, compared to 27.9% and 43.1% by Delta-LSTM. On average, Voyager achieves 73.9% accuracy/coverage, compared with 38.6% for STMS, 43.3% for Domino, 51.1% for ISB, 28.8% for BO, and 52.9% for the Delta-LSTM.

Figure 8 shows that Voyager provides significantly greater IPC improvements than prior art. Normalized to a baseline that has no prefetcher, Voyager improves performance by 41.6%, compared with 14.9% for STMS, 21.7% for Domino, 28.2% for ISB, 13.3% for BO, and 24.6% for Delta-LSTM.

Higher Degree Prefetching. We have so far assumed that all prefetches have a degree of 1, which means that a single prefetch is issued on every trigger access. Coverage can often be improved by increasing the degree to issue multiple prefetches on every trigger address, so we now evaluate Voyager at higher prefetch degrees. To extend Voyager to a degree-k prefetcher, instead of choosing the candidate with the highest probability, we prefetch the top k candidates.

Figure 9 shows that as we increase degree from 1 to 8, Voyager's coverage improves to 65.8%, and it continues to outperform ISB. In fact, we see that Voyager at a degree of 1 outperforms ISB with a degree of 8, which suggests that Voyager can achieve high coverage without being overly aggressive.

Since Voyager can capture both compulsory misses and address correlations, whereas ISB can only capture address correlations, we next compare Voyager with a hybrid of ISB and BO prefetcher [33], which is capable of capturing both compulsory misses and address correlations. The red line in Figure 9 shows that even with a degree of 8, a hybrid of ISB+BO can barely reach the coverage of Voyager with a degree of 1, which again reinforces the observation that Voyager is superior to even the most aggressive versions of prior art. (In the hybrid ISB+BO prefetcher, ISB and BO equally share the available degree, and with a degree of 1, the hybrid falls back to ISB.)

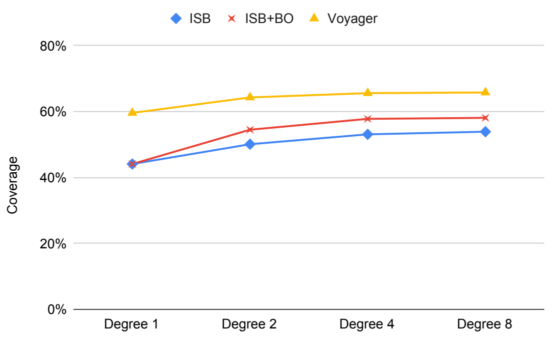


Figure 9: Sensitivity to Prefetch degree.

5.3 Understanding Voyager's Benefits

This section analyzes our results to illustrate the sources of Voyager's benefits. We focus on (1) the memory access patterns that account for Voyager's improved coverage and (2) the effectiveness of Voyager's use of different features and labels.

5.3.1 Access Patterns Breakdown. We first show that Voyager is much more effective than ISB at learning temporal correlations and that Voyager is able to learn a wide variety of temporal access patterns. To isolate Voyager's benefits for temporal access patterns, we first create a crippled version of Voyager that cannot prefetch compulsory misses and is directly comparable to ISB; this version of Voyager does not include deltas in its vocabulary, and we call it Voyager w/o delta. We find that Voyager w/o delta achieves 19.4% better coverage than ISB, confirming that Voyager is more effective at learning temporal correlations than ISB.

We further classify the coverage for both prefetchers into spatial and non-spatial patterns and show that Voyager is better than ISB for both spatial and non-spatial patterns. A prefetch candidate is considered to be spatial if the distance between the last address and the prefetched address is less than a certain threshold (256 cache lines [32]). Figures 10 and 11 show that compared to ISB, Voyager w/o delta improves the prediction of spatial patterns from 45.2% to 56.8%, and it improves the prediction of non-spatial patterns from 13.1% to 22.2%.

Finally, to understand uncovered cases, we further classify the uncovered patterns of Voyager w/o delta and ISB into several categories: (1) *uncovered spatial* refers to spatial patterns that are not covered, (2) *uncovered co-occurrence-k* refers to non-spatial patterns whose addresses co-occur most commonly (we track the top 10 common occurrences), (3) *uncovered others* refers to the remaining less frequent non-spatial patterns, and (4) *uncovered compulsory* misses. Not surprisingly, Voyager w/o delta reduces the percentage of all types of uncovered patterns except compulsory misses.

Of course, compulsory misses are important for benchmarks with large memory footprints. Since machine learning frameworks are flexible, Voyager can easily include the 10 most frequent deltas into the vocabulary, which on *mcf* reduces the percentage of compulsory misses from 21.6% to 0.2%, improving the overall coverage from 49.1% to 68%.

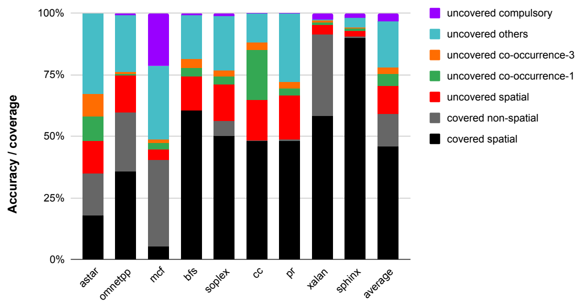


Figure 10: Breakdown of the patterns of ISB.

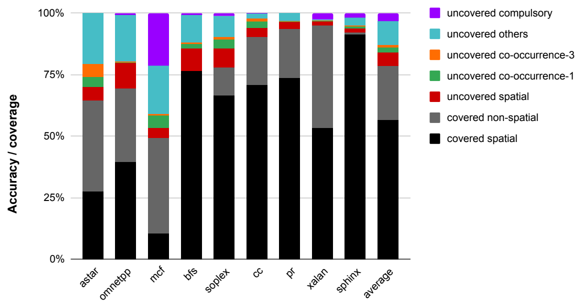


Figure 11: Breakdown of the patterns of Voyager w/o delta.

5.3.2 Features and Labels. Voyager improves coverage and accuracy by introducing new features and a multi-label training scheme. This section dives deeper into these two aspects and provides code examples to illustrate the benefit of each of these two components.

Features. Compared to prior hardware prefetchers, such as STMS and ISB, which use a single data address as a feature, Voyager' neural model utilizes a sequence of data addresses as features. We isolate the effectiveness of Voyager's new features by fixing the labeling scheme: We compare STMS against a version of Voyager that uses only the next address in the global stream as the label, which we refer to as Voyager-global, and we compare ISB against a version of Voyager that uses only the next address of the current PC as the label; we refer to this version as Voyager-PC.

Figure 12 shows that Voyager-global improves coverage over STMS by 19.8%, and Voyager-PC improves coverage over ISB by 16.4%. The right two bars represent two versions of Voyager-PC, one that uses the PC history as a feature and one that does not. We see that unlike with in branch prediction [22, 49, 58] and cache replacement [43], control flow does not help prefetching. Thus, we conclude that for prefetching, the PC is not a useful *feature*. However, as we will see shortly, the PC *is* useful for labeling.

Figures 13 and 14 show concrete code examples that demonstrate the benefit of utilizing the data address history as a feature. The

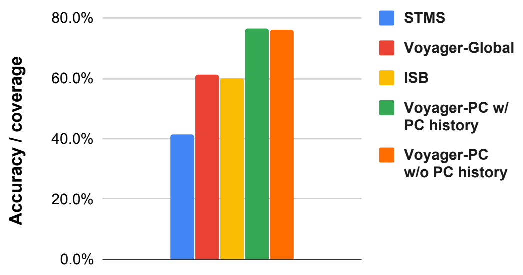
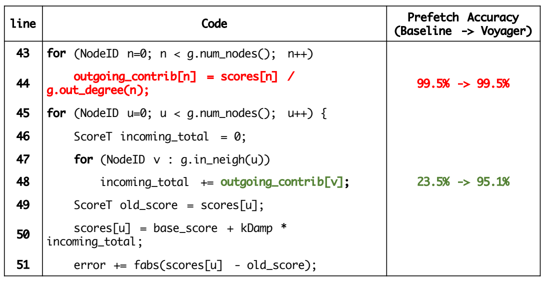


Figure 12: Comparison of different features. Voyager benefits from using data address history as a feature.



line	Code	Prefetch Accuracy (Baseline -> Voyager)
43	`for (NodeID n=0; n < g.num_nodes(); n++)`	
44	`outgoing_contrib[n] = scores[n] / g.out_degree(n);`	99.5% -> 99.5%
45	`for (NodeID u=0; u < g.num_nodes(); u++) {`	
46	`ScoreT incoming_total = 0;`	
47	`for (NodeID v : g.in_neigh(u))`	
48	`incoming_total += outgoing_contrib[v];`	23.5% -> 95.1%
49	`ScoreT old_score = scores[u];`	
50	`scores[u] = base_score + kDamp * incoming_total;`	
51	`error += fabs(scores[u] - old_score);`	

Figure 13: Code example from PageRank.

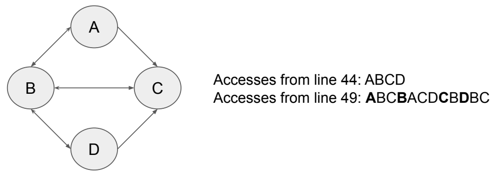

A
B
C
D
Accesses from line 44: ABCD
Accesses from line 49: ABC**B**ACDC**B**D**B**C

Figure 14: An example input graph to PageRank.

code is from the GAP benchmark PageRank, which takes graph-structured inputs. Two loads appear in lines 44 and 48. The load in line 44 is easy to predict since it simply traverses all nodes, or ABCD in the example input graph. Line 48 is more complex, as it traverses all neighbors of all nodes, where each node can be a neighbor of many other nodes. Thus, the next node to be accessed depends on both the current neighbor node and the current neighbor's parent node (shown in bold in Figure 14). Thus, the prediction of the next access by line 48's load becomes more challenging since the notion of a parent node does not exist from the hardware perspective. For example, depending on the parent node, node B can be followed by any other node, which confuses existing temporal prefetchers that only look at one or two past data addresses. Voyager, however, accurately prefetches line 48's load, since it learns to recognize the important sequence of neighbor nodes, which in this case goes through the parent node.

Labeling. As explained in Section 4.4, Voyager is trained with multiple labels, and it is designed to automatically pick the most predictable of these labels. We now evaluate the benefit of this multi-label training scheme. The first five bars in Figure 15 show Voyager's

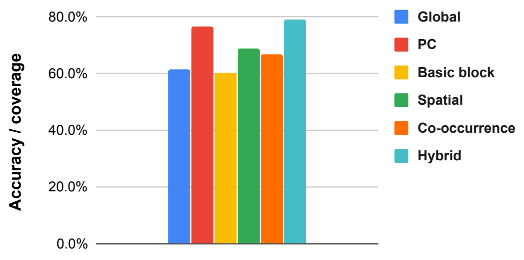
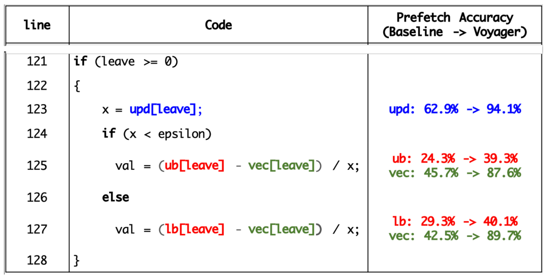


Figure 15: Comparison of different labeling schemes.

unified accuracy/coverage if it were to use a single labeling scheme, and the last bar shows the unified accuracy/coverage with the multi-labeling scheme. We see that on average, the multi-labeling scheme provides a small benefit.

However, we find that different individual benchmarks prefer different labeling schemes. A code example from *soplex* in the SPEC06 benchmark suite, shown in Figure 16, illustrates this point. The lines the precede the code snippet compute the value of the *leave* variable, which if greater than 0 is used in the code snippet to index the arrays *upd*, *ub*, *lb* and *vec*. Voyager prefetches the load of *upd*, *ub* and *lb* by learning from the data address sequence with PC localization. One particularly interesting pattern corresponds to *vec* in lines 125 and 127. *vec[leave]* will be accessed regardless of whether the branch is actually taken, but it will be accessed by one of the two PCs (line 125 or 127), depending on the outcome of the branch. From the perspective of either individual PC, the access to *vec* is hard to predict, since the pattern is shared across the two different PCs. However, our co-occurrence labeling scheme correlates *vec[leave]* with *upd[leave]*, since it is always accessed after *upd[leave]*. This correlation makes the pattern more predictable, so by going beyond PC-localization, Voyager significantly improves upon the baseline by prefetching *vec[leave]* at the point of *upd[leave]*.



line	Code	Prefetch Accuracy (Baseline -> Voyager)
121	`if (leave >= 0)`	
122	`{`	
123	`x = upd[leave];`	upd: 62.9% -> 94.1%
124	`if (x < epsilon)`	
125	`val = (ub[leave] - vec[leave]) / x;`	ub: 24.3% -> 39.3% vec: 45.7% -> 87.6%
126	`else`	
127	`val = (lb[leave] - vec[leave]) / x;`	lb: 29.3% -> 40.1% vec: 42.5% -> 89.7%
128	`}`	

Figure 16: Code example from Soplex.

5.4 Model Compression and Overhead

Compared to the Delta-LSTM prefetcher [13], Voyager's hierarchical representation yields significant storage and computational efficiency. In particular, Voyager reduces the training overhead by 15.1× and prediction latency by 15.6×. At 18,000 nanoseconds per prediction, Voyager's prediction latency is too slow for hardware prediction. We expect that this latency can be reduce by 15× [30]

by avoiding the invocation overhead of Tensorflow's Python front-end, but other techniques will be needed before the latencies are practical.

Voyager also enjoys a dramatically lower storage overhead than Delta-LSTM because of its hierarchical structure. Since the storage cost for neural-prefetchers is dominated by the embedding layer, Voyager's hierarchical structure makes it 20-56× smaller than Delta-LSTM.

To further reduce the storage of Voyager, we can apply standard pruning and quantization methods of Tensorflow [1]. In particular, we find that 80% of Voyager's weights can be pruned, leading to an additional compression of 5-7×. Quantization from 32 bits to 8 bits can provide another 4× compression. Together, these changes result in minimal accuracy loss (less than 1%), and they allow Voyager's storage cost to be 110-200× smaller than that of Delta-LSTM. Significantly, after these optimizations, Voyager is 5-10× smaller than conventional temporal prefetchers, such as STMS, Domino, and ISB.

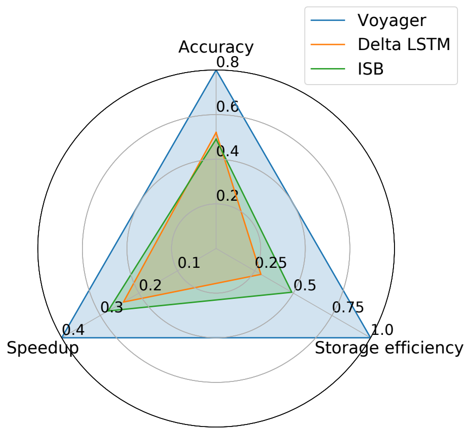


Figure 17: Voyager wins on accuracy, speedup, and storage efficiency. Here storage efficiency is log-scaled and defined as $\frac{1}{1+log_{10}(storage)}$

.

To summarize, Figure 17 shows that with these space optimizations, Voyager outperforms ISB and Delta-LSTM along multiple dimensions.

5.5 Paths to Practicality

While Voyager is not practical, we see three possibles paths that might lead to an eventual practical prefetcher.

Neural-Inspired Practical Prefetchers. Future work could use insights gained from Voyager—and subsequent deep learning research—to build a practical prefetcher that is not based on neural networks. For example, for cache replacement, Glider [43] illustrates that LSTMs can be replaced by perceptrons to advance the state-of-the-art. For data prefetching, the task is more challenging but the potential performance benefit is much greater.

One example of such an insight is that for *mcf* and *search*, two of our hardest-to-predict benchmarks, temporal prefetching provides context that improves delta prefetching. This observation indicates that there is benefit in studying the closer interaction between temporal and spatial prefetching beyond simple hybridization (and in a more equal fashion than was done by the STeMS prefetcher [46]). A second insight is our profitable use of a history of data addresses as a feature, which can inform the feature selection of future hardware prefetchers.

Moreover, our *search* and *ads* results show that there are important workloads for which existing prefetchers perform poorly. The fact that Voyager can do well on these workloads suggests that the community should pay more attention to these Online Transaction Processing workloads.

Profile-Driven Training with Online Inference. The training costs of Voyager can be managed by training the neural model offline during a profiling pass. The weights of the trained model can then be communicated to the hardware with a new ISA interface. The trained model can be used for online inference via a light-weight dedicated hardware block for neural network inference. Zangeneh et al., recently used such an approach to improve branch prediction accuracy using CNNs [58].

Completely Online Neural Prefetchers. In the longer term, we believe that the computational costs of ML will improve with techniques such as few-shot learning and hierarchical softmax, which trade off some accuracy for dramatic efficiency gains. For example, few shot learning reduces the size of training data by 20-80×. We estimate that hierarchical softmax will reduce both training and inference time by 3-4× by further reducing the number of classes. Related work also shows that by implementing neural networks in languages such C++ instead of Tensorflow's Python interface, performance can be improved by 15× [30]. Efforts such as these can eventually lead to neural prefetchers that have moderate computation overheads for both training and inference.

6 CONCLUSIONS

In this paper, we have created a probabilistic model of data prefetching in terms of features and localization, and we have presented a new neural model of data prefetching that accommodates both delta patterns and address correlation. The key to accommodating address correlation is our hierarchical treatment of data addresses: We separate the addresses into pages and offsets, and our model makes predictions for them jointly. Our neural model shows that significant headroom remains for data prefetchers. For a set of irregular SPEC and graph benchmarks, Voyager achieves 79.6% coverage and improves IPC over a baseline with no prefetching by 41.6%, compared with 57.9% and 28.2%, respectively, for an idealized ISB prefetcher. We also present results for two important commercial programs, Google's *search* and *ads*, which until now have seen little benefit from any data prefetcher. Voyager gets 37.8% coverage for *search* (13.8% for ISB) and 57.5% for *ads* (26.2% for ISB).

Work remains in further reducing the computational costs of our neural prefetcher, but our analysis reveals some interesting insights about temporal prefetching. For example, we find that a long data address history serves as a good feature to predict irregular accesses, and we find that multiple localizers provide significant benefits for some hard-to-predict benchmarks. Thus, even if literal neural models remain impractical, we hope that these insights will guide the development of practical prefetchers in terms of features and localization schemes.

ACKNOWLEDGMENTS

We thank our shepherd, Mingyu Gao, the anonymous referees, and Ashish Naik for their helpful comments on earlier versions of this paper. We thank the Tools, Architecture, and Optimization team at Google for collecting traces of Google applications. This work was funded in part by a Google Research Award, a gift from Arm Research, NSF Grant CCF-1823546, and a gift from Intel Corporation through the NSF/Intel Partnership on Foundational Microarchitecture Research.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 265–283, 2016.

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

[3] Mohammad Bakhshalipour, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. Domino temporal data prefetcher. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 131–142, 2018.

[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, 2019.

[5] Scott Beamer, Krste Asanović, and David Patterson. The GAP benchmark suite. *arXiv preprint arXiv:1508.03619*, 2015.

[6] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pages 265–275. IEEE, 2003.

[7] Doug Burger, Thomas R. Puzak, Wei-Fen Lin, and Steven K. Reinhardt. Filtering superfluous prefetches using density vectors. In *Proceedings of the International Conference on Computer Design: VLSI in Computers & Processors (ICCD)*, pages 124–133, 2001.

[8] Chi F. Chen, Se-Hyun Yang, Babak Falsafi, and Andreas Moshovos. Accurate and complexity-effective spatial pattern prediction. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture (HPCA)*, pages 276–288, 2004.

[9] Trishul M. Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 191–202, 2001.

[10] Yuan Chou. Low-cost epoch-based correlation prefetching for commercial applications. In *Proceedings of the 40th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pages 301–313, 2007.

[11] Keith I. Farkas, Paul Chow, Norman P. Jouppi, and Zvonko Vranesic. Memory-system design considerations for dynamically-scheduled processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA)*, pages 133–143, 1997.

[12] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. Simpoint 3.0: Faster and more flexible program phase analysis. *Journal of Instruction Level Parallelism*, 7(4):1–28, 2005.

[13] Milad Hashemi, Kevin Swersky, Jamie A Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. Learning memory access patterns. *arXiv preprint arXiv:1803.02329*, 2018.

[14] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[15] Zhigang Hu, Margaret Martonosi, and Stefanos Kaxiras. TCP: tag correlating prefetchers. In *International Symposium on, High Performance Computer Architecture (HPCA)*, pages 317–326, 2003.

[16] Ibrahim Hur and Calvin Lin. Memory prefetching using adaptive stream detection. In *Proceedings of the 39th International Symposium on Microarchitecture (MICRO)*, pages 397–408, 2006.

[17] Yasuo Ishii, Mary Inaba, and Kei Hiraki. Access map pattern matching for high performance data cache prefetch. *Journal of Instruction-Level Parallelism*, 13:1–24, 2011.

[18] Robert A. Jacobs, Michael I. Jordan, Steven J. Nowlan, and Geoffrey E. Hinton. Adaptive mixtures of local experts. *Neural computation*, 3(1):79–87, 1991.

[19] 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 (MICRO)*, pages 247–259, 2013.

[20] Akanksha Jain and Calvin Lin. Back to the future: Leveraging belady's algorithm for improved cache replacement. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2016.

[21] Aamer Jaleel, Robert S Cohn, Chi-Keung Luk, and Bruce Jacob. Cmp$im: A Pin-based on-the-fly multi-core cache simulator. In *Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS), co-located with ISCA*, pages 28–36, 2008.

[22] Daniel A Jiménez. Multiperspective perceptron predictor. In *The Journal of Instruction-Level Parallelism 5th JILP Workshop on Computer Architecture Competitions (JWAC-5), Championship Branch Prediction, (co-located with ISCA 2016)*, 2016.

[23] Daniel A Jiménez and Calvin Lin. Dynamic branch prediction with perceptrons. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture (HPCA)*, pages 197–206, 2001.

[24] Daniel A Jiménez and Elvira Teran. Multiperspective reuse prediction. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 436–448. IEEE, 2017.

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

[26] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA)*, pages 252–263, 1997.

[27] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *International Symposium on Computer Architecture (ISCA)*, pages 364–373, 1990.

[28] Samira Khan, Yingying Tian, and Daniel A Jiménez. Sampling dead block prediction for last-level caches. In *43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 175–186, 2010.

[29] Jinchun Kim, Seth H Pugsley, Paul V Gratz, AL Reddy, Chris Wilkerson, and Zeshan Chishti. Path confidence based lookahead prefetching. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, page 60. IEEE Press, 2016.

[30] Tim Kraska, Alex Beutel, Ed H. Chi, Jeff Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*, 2018.

[31] Sanjeev Kumar and Christopher Wilkerson. Exploiting spatial locality in data caches using spatial footprints. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 357–368, 1998.

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

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

[34] Jinseok Nam, Jungi Kim, Eneldo Loza Mencía, Iryna Gurevych, and Johannes Fürnkranz. Large-scale multi-label text classification-revisiting neural networks. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 437–452, 2014.

[35] Kyle J. Nesbit, Ashutosh S. Dhodapkar, and James E. Smith. AC/DC: an adaptive data cache prefetcher. In *13th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 135–145, 2004.

[36] Kyle J. Nesbit and James E. Smith. Data cache prefetching using a global history buffer. *IEEE Micro*, 25(1):90–97, 2005.

[37] Subbarao Palacharla and Richard E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 24–33, April 1994.

[38] Leeor Peled, Shie Mannor, Uri Weiser, and Yoav Etsion. Semantic locality and context-based prefetching using reinforcement learning. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 285–297, 2015.

[39] Leeor Peled, Uri Weiser, and Yoav Etsion. A neural network prefetcher for arbitrary memory access patterns. *ACM Transactions on Architecture and Code Optimization (TACO)*, page 37, 2019.

[40] 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 *IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[41] Suleyman Sair, Timothy Sherwood, and Brad Calder. A decoupled predictor-directed stream prefetching architecture. *IEEE Transactions on Computers*, 52(3):260–276, March 2003.

[42] Manjunath Shevgoor, Sahil Koladiya, Rajeev Balasubramonian, Chris Wilkerson, Seth H. Pugsley, and Zeshan Chisthi. Efficiently prefetching complex address patterns. In *Proceedings of the 48th International Symposium on Microarchitecture*

(MICRO), pages 141–152, 2015.

[43] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. Applying deep learning to the cache replacement problem. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 413–425, 2019.

[44] A.J. Smith. Sequential program prefetching in memory hierarchies. *IEEE Transactions on Computers*, 11(12):7–12, December 1978.

[45] Yan Solihin, Jaejin Lee, and Josep Torrellas. Using a user-level memory thread for correlation prefetching. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, pages 171–182, 2002.

[46] Stephen Somogyi, Thomas F. Wenisch, Anastasia Ailamaki, and Babak Falsafi. Spatio-temporal memory streaming. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 69–80, 2009.

[47] Stephen Somogyi, Thomas F. Wenisch, Anastassia Ailamaki, Babak Falsafi, and Andreas Moshovos. Spatial memory streaming. In *Proceedings of the 33th Annual International Symposium on Computer Architecture (ISCA)*, pages 252–263, 2006.

[48] Ajitesh Srivastava, Angelos Lazaris, Benjamin Brooks, Rajgopal Kannan, and Viktor K. Prasanna. Predicting memory accesses: The road to compact ml-driven prefetcher. In *Proceedings of the International Symposium on Memory Systems (MEMSYS)*, pages 461–470, 2019.

[49] Stephen J Tarsa, Chit-Kwan Lin, Gokce Keskin, Gautham Chinya, and Hong Wang. Improving branch prediction by modeling global history with convolutional neural networks. *arXiv preprint arXiv:1906.09889*, 2019.

[50] Elvira Teran, Zhe Wang, and Daniel A Jiménez. Perceptron learning for reuse prediction. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016.

[51] Grigorios Tsoumakas and Ioannis Katakis. Multi-label classification: An overview. *International Journal of Data Warehousing and Mining (IJDWM)*, 3(3):1–13, 2007.

[52] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.

[53] Thomas F. Wenisch, Michael Ferdman, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. Temporal streams in commercial server applications. In *IEEE International Symposium on Workload Characterization*, pages 99–108, 2008.

[54] Thomas F Wenisch, Michael Ferdman, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. Practical off-chip meta-data for temporal memory streaming. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture (HPCA)*, pages 79–90, 2009.

[55] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C. Steely, Jr., and Joel Emer. SHiP: Signature-based hit predictor for high performance caching. In *44th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 430–441, 2011.

[56] Hao Wu, Krishnendra Nathella, Joseph Pusdesris, Dam Sunwoo, Akanksha Jain, and Calvin Lin. Temporal prefetching without the off-chip metadata. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 996–1008, 2019.

[57] Hao Wu, Krishnendra Nathella, Dam Sunwoo, Akanksha Jain, and Calvin Lin. Efficient metadata management for irregular data prefetching. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*, pages 449–461, 2019.

[58] Siavash Zangeneh, Stephen Pruett, Sangkug Lym, and Yale N Patt. Branchnet: A convolutional neural network to predict hard-to-predict branches. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 118–130, 2020.