Predicting Memory Accesses: The Road to Compact ML-driven Prefetcher

Ajitesh Srivastava
University of Southern California
USA
ajiteshs@usc.edu

Angelos Lazaris
University of Southern California
USA
alazaris@usc.edu

Benjamin Brooks
University of Southern California
USA
bjbrooks@usc.edu

Rajgopal Kannan
Army Research Lab - West
USA
rajgopal.kannan.civ@mail.mil

Viktor K. Prasanna
University of Southern California
USA
prasanna@usc.edu

ABSTRACT

With the advent of fast processors, TPUs, accelerators, and heterogeneous architectures, computation is no longer the only bottleneck. In fact for many applications, speed of execution is limited by memory performance. To address memory performance, more accurate prefetching is necessary. While sophisticated machine learning algorithms have shown to predict memory accesses with high accuracy, they suffer with several issues that prevent them from being practical solutions as hardware prefetchers. These issues are centered around size of the model that results in high memory requirement, high latency and difficulty in online retraining. As the first step towards building ML-based prefetchers, we propose a compressed-LSTM approach for accurate memory access prediction. With a novel compression technique based on output encoding, we show that for the problem of predicting one of n memory locations, our technique results in $O(n/\log n)$ compression factor over the traditional LSTM approach. We further demonstrate through experiments on several benchmarks that the prediction accuracy drop due to compression is small and the training is fast. The actual compression obtained is of the order of 100×.

CCS CONCEPTS

• Information systems → Data mining; **• Computer systems organization → Neural networks**; **• Software and its engineering →** *Memory management*.

KEYWORDS

memory access prediction, prefetching, deep learning, compression

ACM Reference Format:
Ajitesh Srivastava, Angelos Lazaris, Benjamin Brooks, Rajgopal Kannan, and Viktor K. Prasanna. 2019. Predicting Memory Accesses: The Road to Compact ML-driven Prefetcher. In *Proceedings of the International Symposium on Memory Systems (MEMSYS '19), September 30-October 3, 2019, Washington, DC, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3357526.3357549

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

MEMSYS '19, September 30-October 3, 2019, Washington, DC, USA

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-7206-0/19/09…$15.00
https://doi.org/10.1145/3357526.3357549

1 INTRODUCTION

Improving instructions per cycle (IPC) for single-thread applications with stagnating clock frequency requires dealing with fundamentally difficult constraints, e.g., branch mispredictions and cache misses. Overcoming these constraints, while difficult, promises to bring substantial increases in IPC. This has now become a necessity to support the scale of growing data. Fast processors, TPUs, accelerators, and heterogeneous architectures, have enabled fast computation due to which memory performance has become the bottleneck in many applications. Many applications are memory bound and the problem of reducing the latency of memory accesses must be addressed. Several emerging memory technologies such 3D-Stacked DRAM and Non-volatile Memory attempt to address memory bottleneck issues from a hardware perspective, but with a tradeoff among bandwidth, power, latency, and cost.

Rather than redesigning existing algorithms to suit specific memory technology, we propose to develop Machine Learning (ML) based approach that automatically learns access patterns which may be used to prefetch data. Specifically, LSTM (Long-Short Term Memory) based Deep Learning has been successfully used in natural language tasks such as part of speech tagging [1], grammar learning [2], and text prediction [3]. In fact, the "Quicktype" function of iPhone uses LSTM [4] to predict the next word while typing. Since memory accesses have an underlying grammar similar to natural language (albeit simpler due to being context free), such models are naturally applicable to learning accesses. However, only recently, application of LSTM has gained the attention of researchers for learning accesses [5, 6]. While it has been shown to obtain high precision and recall, the approach of training offline and testing online for individual application is not a practical prefetcher, as pointed by the researchers themselves, and is only the initial step towards building one.

A practical LSTM based prefetcher implementation requires dealing with certain challenges that we address (i) size of the machine learning model (small - to enable fast inference and ensure feasibility of implementation); (ii) training with large traces to obtain a (small) model that is highly accurate in predicting memory accesses; (iii) ensuring real-time inference; and (iv) **retraining** the model online, on-demand to learn application specific models, which would

require fast learning with small amount of data. A typical size (number of parameters) of LSTM model for memory access prediction is dominated by the output layer due to a dense layer connecting tens of thousands of outputs (possible memory addresses). This forms the basis of all the challenges listed earlier. We propose a compression technique that achieves a factor of $O(n/\log n)$ in reduction of parameters without compromising accuracy, thus overcoming these challenges. Note that our objective, in this paper, is not to develop a full scale prefetcher, but to design a highly accurate and compact LSTM based access prediction model that demonstrates the utility of highly compressed LSTMs for improving prefetching. A prefetcher built on top of this model and its hardware implementation will be explored in future work. Specifically, our contributions are as follows.

- We propose an LSTM-based approach to predict the next access with high accuracy.
- We propose a compression technique that results in a compact design with number of parameters reduced by a theoretical factor of $\approx n/\log n$.
- We present experiments on several benchmarks to show that our compressed LSTM (with parameter reduction of the order of 100×) can reach high accuracy comparable to the original LSTM model.
- We demonstrate that our compressed LSTM can be retrained only in few epochs and can reach high accuracy using a small trace.
- We show that compressed LSTMs have the potential to significantly improve prefetching. We use a simple cache simulator that prefetches blocks based on access predictions made by our compressed LSTM and comparatively evaluate performance metrics versus several state-of-the-art prefetchers.

2 RELATED WORK

The problem of leveraging memory access patterns to improve prefetching has been studied in the literature. [7] and [8] present the first attempts to leverage predictions using a) a Markov predictor, b) a linear predictor, and c) a time delay neural network, in order to dynamically reconfigure interconnection networks for hiding processor-memory control latency. The authors use three benchmark apps and also conclude that neural network approaches are computationally expensive, and thus not tractable, something that is not true nowadays. Similarly, feedforward neural network designs in [9] have barely outperformed classical benchmarks (LRU, ARC, etc.) in caching policy while requiring excess training time. [10] leverages a Markov chain model that is able to predict access patterns and detect anomalous behaviors for a given app, by essentially detecting deviations from the expected behavior. The work in [11] combined a linear model for prefetching with a reinforcement learning approach to eviction choice. The work in [12] presents a similar approach that uses a classification framework to detect malware by analyzing its access patterns. The remaining of the relevant work focuses on leveraging predictions to improve prefetching [5, 6, 13–15], which is different than our proposed framework in the sense that the prediction error is calculated only on cache misses (or overall speedup) and not on specific memory location prediction misses (which is more fine-grained prediction). In addition, our fine-grained modeling framework can enable memory layout optimizations. The most relevant approach to our work is presented in [6] where the authors propose a framework that uses LSTMs to improve prefetching. A similar LSTM approach for prefetching is presented in [5]. In [14], the authors propose the use of logistic regression, and decision tree models to enhance prefetching. The authors in [15] evaluate various machine learning models on their ability to improve prefetching for data center applications. Neural networks and decision trees where shown to achieve the highest performance in this application domain. The work in [13], [16], and [17] presents an extensive evaluation of LSTM for prefetching, achieving similar performance improvements as the other LSTM based approaches.

Distinguishing from Existing Work. The LSTM based approach in [6] is the most recent and promising work in this area, however, the authors approach the problem from an offline perspective. The approach is impractical to be directly applied for prefetching, and as stated by the authors, is only a first step towards an LSTM-based prefetcher. They, and several state-of-the-art machine learning based access predictors perform the training on cache misses as it reduces the size of training. However, an accurate prefetcher will change the the distribution of cache misses and hence invalidate its own trained model. Secondly, to achieve higher accuracy, some online training is necessary to learn application specific patterns. We propose to use a hybrid offline+online training approach where a base model is trained offline first. At runtime, in case of low accuracy, a more specialized model is trained in real-time with the hypotheses that high accuracy can be obtained by only few training samples with few epochs, and that this high accuracy can be sustained for a long period of time before another round of retraining is required.

While the focus of this paper is on improved memory access prediction as a first step towards building ML based prefetchers in emerging architectures (Sec. 5.2), there is also an extensive body of work on non deep learning based prefetching. Examples of such prefetchers include the Best-Offset Prefetcher (BOP) [18], the Signature Path Prefetcher (SPP) [19], the Variable Length Delta Prefetcher (VLDP) [20], and the Composite Prefetcher (TPC) [21]. These are state-of-the-art optimized prefetchers which can also leverage various hardware and architecture optimizations to deliver further performance gains. [18] prefetches the address corresponding to the most popular delta (address offset) found in training. [20] memorizes the deltas that follow access sequences of varying lengths. [19] utilizes a custom signature - defined as a function of memory address delta and previous signature, to output the next predicted delta. [21] selectively applies several prefetcher designs to more accurately recognize unique access patterns. In order to demonstrate the potential benefits of LSTM based prefetching, we use a simple cache simulator to compare the performance of our access prediction against state-of-the-art prefetchers (Sec. 4.6).

3 APPROACH

In order to better understand the memory access patterns for different applications, we analyze six long traces generated using the PARSEC benchmark suite [22], and study their autocorrelation coefficients (ACFs) for various lags, as well as the frequency distribution

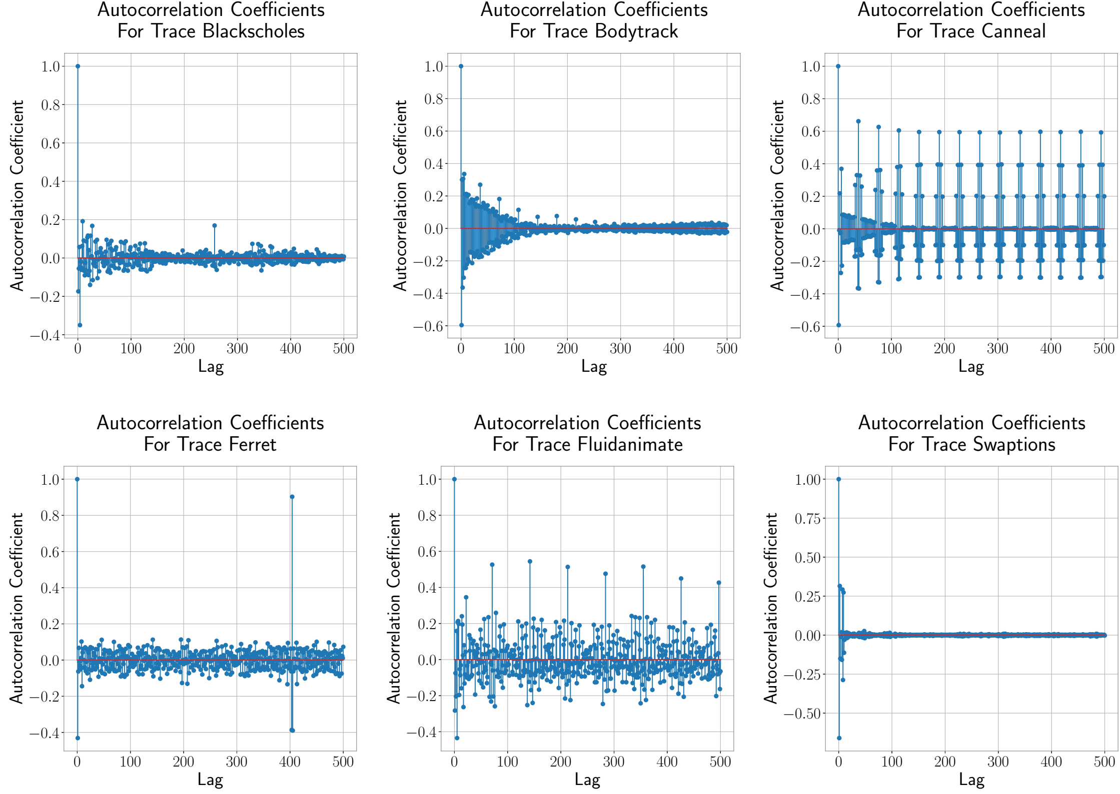

Figure 1: Autocorrelation coefficients for each trace for various lags.

of the memory deltas, i.e. the difference between consecutive memory addressees accessed. As we can see in Fig. 4, the ACFs reveal the existence of Long Range Dependencies (LRD) of various lengths, especially in the case of Canneal, Ferret, and Fluidanimate traces, whereas the Blackscholes, Bodytrack, and Swaptions traces appear to have ACFs that decay faster. Nevertheless, these patterns are good indicators for the suitability of LSTMs to model memory access patterns since LSTMs were proposed to handle the challenges imposed by LRDs [23].

3.1 LSTM-based Prediction

An LSTM model is a form of a recurrent neural network that has gained popularity in the recent years due to its effectiveness in modeling complex time-series with time lags of unknown size that separate important events [23, 24]. The main idea of LSTM is the use of self-loops where the gradient can flow for long durations without vanishing or exploding. This, in combination with the use of a forget-gate, allows the LSTM to accumulate knowledge that can be "forgotten" later depending on the input data.

LSTMs are characterized by the following recursive equations:

$$\mathbf{f}^{(t)} = \sigma\left(\mathbf{W}_f \mathbf{x}^{(t)} + \mathbf{U}_f \mathbf{h}^{(t-1)} + \mathbf{b}_f\right) \tag{1}$$

$$\mathbf{i}^{(t)} = \sigma\left(\mathbf{W}_i \mathbf{x}^{(t)} + \mathbf{U}_i \mathbf{h}^{(t-1)} + \mathbf{b}_i\right) \tag{2}$$

$$\tilde{\mathbf{c}}^{(t)} = \tanh\left(\mathbf{W}_c \mathbf{x}^{(t)} + \mathbf{U}_c \mathbf{h}^{(t-1)} + \mathbf{b}_c\right) \tag{3}$$

$$\mathbf{c}^{(t)} = \mathbf{i}^{(t)} \odot \tilde{\mathbf{c}}_{(t)} + \mathbf{f}^{(t)} \odot \mathbf{c}^{(t-1)} \tag{4}$$

$$\mathbf{o}^{(t)} = \sigma\left(\mathbf{W}_o \mathbf{x}^{(t)} + \mathbf{U}_o \mathbf{h}^{(t-1)} + \mathbf{b}_o\right) \tag{5}$$

$$\mathbf{h}^{(t)} = \mathbf{o}^{(t)} \odot \tanh(\mathbf{c}^{(t)}) \tag{6}$$

where $\mathbf{f}^{(t)}, \mathbf{i}^{(t)}, \tilde{\mathbf{c}}^{(t)}, \mathbf{c}^{(t)}, \mathbf{o}^{(t)}, \mathbf{h}^{(t)}$ are the forget gate, input gate, candidate state, current state, output gate, and hidden state, respectively, $\mathbf{W}_f, \mathbf{W}_i, \mathbf{W}_c, \mathbf{W}_o$ are the input weights for the forget gate, input gate, candidate state gate, and output gate, respectively, and $\mathbf{U}_f, \mathbf{U}_i, \mathbf{U}_c, \mathbf{U}_o$ are the recurrent weights for the forget gate, input gate, current state, and output gate, respectively. In addition, $\odot$ is the (element-wise) Hadamard product, and σ is the sigmoid function.

3.2 LSTM Based Sequence Prediction

Given a memory trace we wish to train an LSTM model that can accurately predict future accesses. Instead of actual memory locations, we transform the memory traces to sequences of integers using ordinal encoding, and then we transform the resulting sequence

into a sequence of memory deltas by calculating the difference between consecutive addresses, similar to [6]. The reason for this is to allow the model to predict memory locations for any future execution of the same application, since the relative memory differences are expected to stay consistent. We model the problem of memory access prediction as a classification problem instead of regression, similar to [6], where each memory address can be treated as a word of a large vocabulary. In order to have a compact model with adequate levels of accuracy and to avoid overfitting, we remove deltas that have frequency of $\leq N$ (e.g. 10, however the parameter can be adjusted depending on the application, or even automated using a percentile-based threshold).

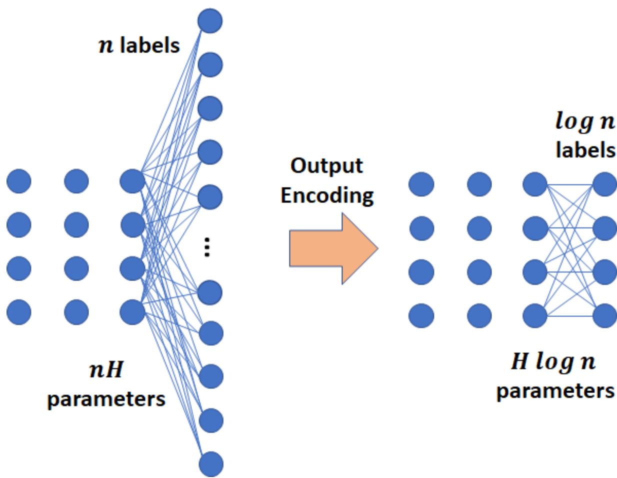

Figure 2: Compression using output encoding.

3.3 Compression

Since the LSTM model is intended to be the basis of a prefetcher, we would like it to be as compact as possible. The size (number of parameters) of the simple LSTM model for classification is dominated by the dense last layer. Few thousands or even hundreds of output nodes may lead to slowing down of inference due to large number of output labels. Suppose, the number of labels is n and the layer before the last one has a dimension H. This results in Hn connections (parameters). Suppose the number of rest of the parameters is H_0 which is typically small due to low dimensional latent representation. Then total number of parameters is $H_0 + Hn$. In our experiments, $n \approx 50,000 \gg H, H_0$. We propose to achieve a very high compression by representing the vocabulary in binary (Figure 2), i.e., for vocabulary of size n, we create the output layer with $\log n$ nodes each of which can take a 0 or 1 value. Now, we train the LSTM to predict a multi-label output with $\log n$ labels instead of a single label (1 out of n) classification problem. Now, the number of parameters in the model is $H_0 + H \log n$. Therefore, compression ratio r is

$$\frac{H_0 + Hn}{H_0 + H \log n} \approx \frac{n}{\log n}. \tag{7}$$

Note that we have made the prediction problem much harder due to the fact that all the bits the $\log n$ bits need to be predicted correctly for the right memory access prediction. However, the compression factor is $O(\frac{n}{\log n})$ which is extremely high. In fact, the experiments demonstrate that order of 1000x compression can be achieved with a negligible loss in accuracy.

Note that depending on the implementation, the input layer would also require a similar compression at a possible cost of accuracy loss. In our experiments 4, we explore the effects of both output compression and double (input+output) compression.

4 EXPERIMENTS

4.1 Setup

In order to evaluate our proposed memory access pattern prediction architecture, we conducted extensive experimentation using the PARSEC benchmark [22]. The Intel Pin [25] tool was used to obtain memory access traces for each application. Instead of actual memory locations, we transform the memory traces to sequences of deltas by subtracting each consecutive hexadecimal memory address and converting the final difference to decimal. The reason for this is to allow the model to predict memory locations for any future execution of the same application, since the relative memory differences are expected to stay consistent, as well will also show experimentally below.

As explained earlier, we approach the memory access prediction as a classification problem where each memory address can be treated as a word from a large vocabulary (i.e. the vocabulary of memory deltas). In order to have a compact model with adequate levels of accuracy and to avoid overfitting, we remove deltas that have frequency less than a certain threshold (e.g. 10). This allows the total vocabulary size to be kept relatively small, without compromising too much on the accuracy. The datasets used are summarized in Table 1.

4.2 Prediction Models

We used several variations of LSTM models for sequence prediction, as well as simpler models that can be used as baselines. Specifically, we implemented the following models:

(1) **Vanilla LSTM (vlstm):** This is a simple LSTM architecture for sequence prediction with an embedding layer with 10 units, followed by an LSTM layer with 50 units, followed by a dense layer with 50 units, and one hot encoded outputs (for the deltas). We also used a dropout of 10%, look back window 3 (i.e., takes last three access predictions as input), 5 training epochs, a batch size 256, a cap of the vocabulary at 50K most frequent deltas (larger vocabularies where not feasible due to the large number of parameters that needed to be trained), and 50-50 train/test split. The loss function used was the categorical cross entropy with softmax activation functions.
(2) **Compressed LSTM (clstm):** This is our compressed-LSTM which is similar in architecture as in 1) above, with the only difference that the output deltas have been converted to 16-bit binary format. This reduces the output size and the neural network is trained to predict binary outputs that are later converted to decimal. For this, we use sigmoid activation function and binary cross entropy loss function.

Table 1: Memory Access Pattern Datasets.*Note:#deltas represents the distinct deltas used in first 200K accesses only, after a 100K accesses of warmup.

dataset	#addresses	#deltas *	# vlstm parameters	# clstm parameters	# dclstm parameters
blackscholes	63,141,878	14,708	279,868	160,096	23,944
bodytrack	67,921,497	65,536	3,062,383	668,376	23,944
canneal	525,279,009	65,536	3,062,383	668,376	23,944
ferret	269,989,952	65,536	3,062,383	668,376	23,944
fluidanimate	838,028,424	56,516	3,062,383	578,176	23,944
swaptions	66,999,281	44,076	2,700,836	453,776	23,944

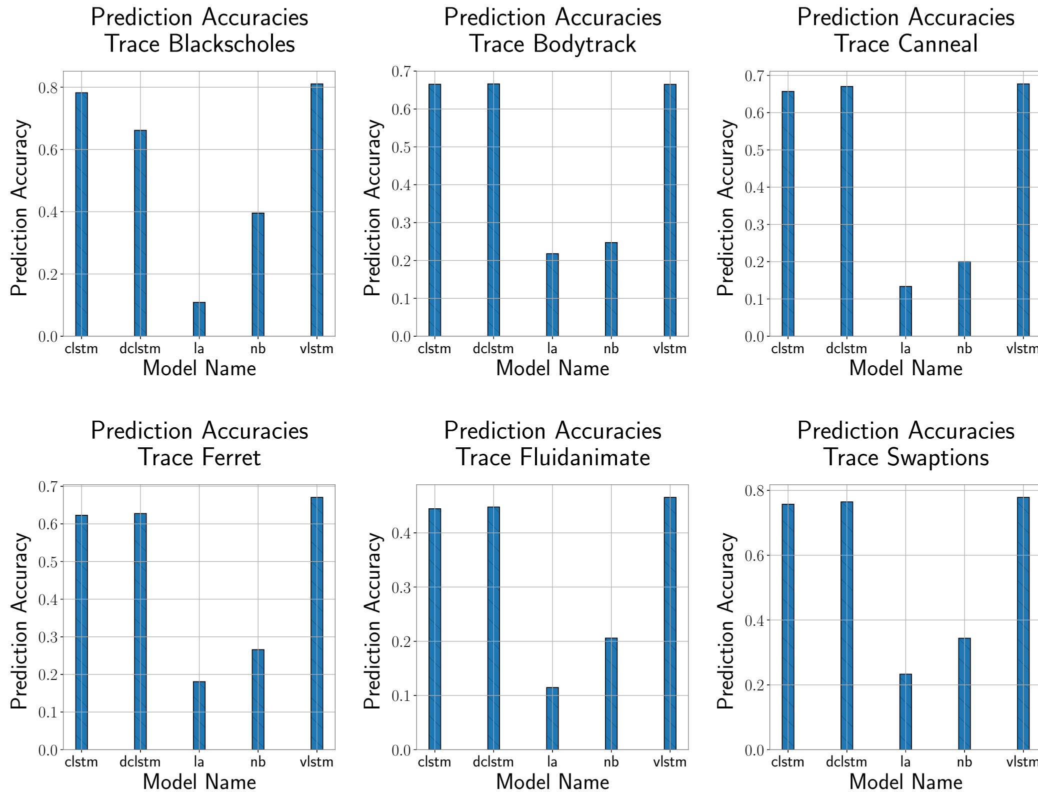

Figure 3: Prediction accuracies for each trace for offline training and predictions on the same trace.

(3) **Doubly Compressed LSTM (dclstm):** This is exactly the same architecture as in 2) above, with the only difference that the input deltas are also converted to 16-bit binary format to reduce the input vocabulary size and reduce the number of model parameters.
(4) **Last Access Prediction (la):** The next memory access delta is predicted to be the exact same as the one directly before it. This is used as the simplest possible baseline.
(5) **Naive Bayes (nb):** This model caps the vocabulary at the 2K most popular deltas (larger vocabularies where impossible to use for Naive Bayes due to memory errors). It also uses the previous four memory access deltas (encoded by the vocabulary) to predict the next delta using a Naive Bayes model structure. The accuracy of the model is calculated only on the deltas seen in order to provide more optimistic estimates, since the 2K vocabulary cap is significantly less than the 65K of CLSTM or the 50K of VLSTM.

Evaluation metric - Accuracy We use accuracy on a test set in each of the scenarios listed below. Accuracy is calculated as the fraction of correctly predicted address. To be considered correct, for "vlstm" and "la", the prediction has to match exactly with the true access. If the true access is outside the set of accesses used to train the model, then it is counted as incorrect. For "clstm" and "dlstms" all the 16 bits must correctly match the binary representation of the true access. We allow a more optimistic accuracy evaluation of "nb" to make the baseline stronger. It should be easier for "nb" to

achieve a higher accuracy as it only has to correctly predict one of 2K accesses compared to 50K for LSTM-based approaches. Also, unlike LSTM-based approaches "nb" is not penalized on an address that is not one of the 2K used to train it.

Scenarios Tested Below we summarize the memory access prediction scenarios tested:

1. **Offline Training/Testing On Same Trace:** In this scenario, we generate a trace which is used for training and testing the model. We discard the first 100K accesses as warmup, and then use the next 200K for training and the next 200K for testing. Other chunks of memory accesses produced similar results, and so we only presenting this offline setting.
2. **Emulating Online Learning**: In this scenario, the model collects few thousands memory accesses (e.g. 10K) and start training the LSTM model. The model can be used to generate predictions right away (e.g. after the first epoch of training), and while more epochs are completed, more accurate estimates can be generated. The average accuracy is calculated in rolling windows of 50K samples, and once the accuracy falls below a threshold, the retraining is performed but starting from the weights of the neural network from the previous training period, so that the network accumulates knowledge. In our case, the threshold is set to 60% since it provided a good trade off between accuracy and frequency of retraining
3. **Offline Training/Testing On Different Traces:** In this scenario, we generate a trace from a given app which is used for training a model. The model produced is later used to predict the memory access patterns of the same app in a later rerun (same input as well as different input), such that we can see how reusable models can be and how feasible it is to have a *general* model that can be used for a large variety of inputs.
4. **Prefetch Evaluation:** The main goal of our work is accurately predicting memory accesses using compressed LSTM models and not demonstrating a full prefetcher, which we plan to build in future work. However, to perform a comparison with state-of-the-art prefetchers, we integrate our compressed LSTM predictions with a cache simulator to demonstrate a preliminary version of the prefetcher.

4.3 Offline Learning

The results for offline training/testing on the same trace are shown in Fig. 3 for each trace. As we can observe, all the variations of the LSTM models perform much better than the baselines (i.e. "nb" and "la"), of which "la" is always the worst. Among the LSTM models, the "vlstm" performs slightly better at the expense of larger number of parameters to train, and longer training times, which where measured to be up to 15x the training times of the "clstm". On the other hand, "clstm" and "dlstm" appear to perform very closely to "vlstm" for all the six apps under study, that shows that we can obtain a consistent compression of space and time and yet not lose too much on accuracy. Regarding the model accuracies overall, for five out of the six apps, the compressed LSTMs achieved accuracies that ranged from 82% to 63% (extremely close to their respective "vlstm"). The most challenging trace appears to be the Fluidanimate where

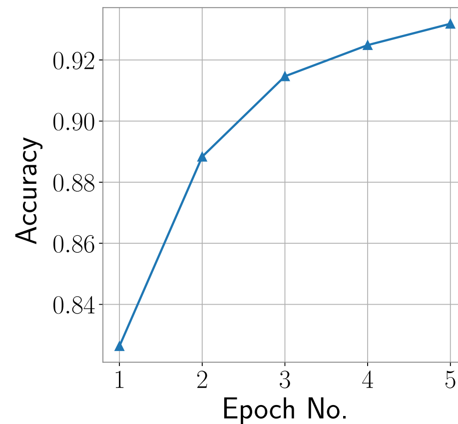

Figure 4: Training accuracy per training epoch for the Swaptions app. Five epochs appear to be enough for the training accuracy to start stabilizing.

LSTMs appear to overfit and could not generalize very well. Of course, these can be treated with more data or further tuning of the model hyperparameters. However, in this paper, we did not fine tune the model to each trace separately but instead we tried to provide a simple model setup that can be used to build a generic predictor that can work for various apps. Details of compression can be found in Table 1. Note that "clstm" results in up to 5× compression, while "dlstm" leads to $> 100\times$ compression.

4.4 Emulating Online Learning

The results for the online learning models are shown in Fig. 5 from where we can see that each LSTM model achieves a steady testing accuracy over time that is very close to their offline averages. The only exception is for Fluidanimate, where the initial accuracy was really good (around 80%) but due to overfitting, the model could not predict the subsequent set of deltas very well. However, after two retraining epochs (50K accesses each), the model performance went back to its initial levels.

4.5 Testing on Different Traces

The results for pretrained models on reruns are shown in Fig. 6. We again observe that the LSTM models outperform the baselines in predicting accesses on both (i) the same input ("rerun") and (ii) different input ("new"). Again, we observe that the compressed LSTM models achieve comparable accuracy to "vlstm". The only significant accuracy drop is observed in Blackscholes for"dclstm" for rerun on the same input ($\approx 20\%$). We believe that further tuning of the embedding layer hyperparameters can fix this issue, as that is the only distinction from "clstm"which does not show a significant drop.

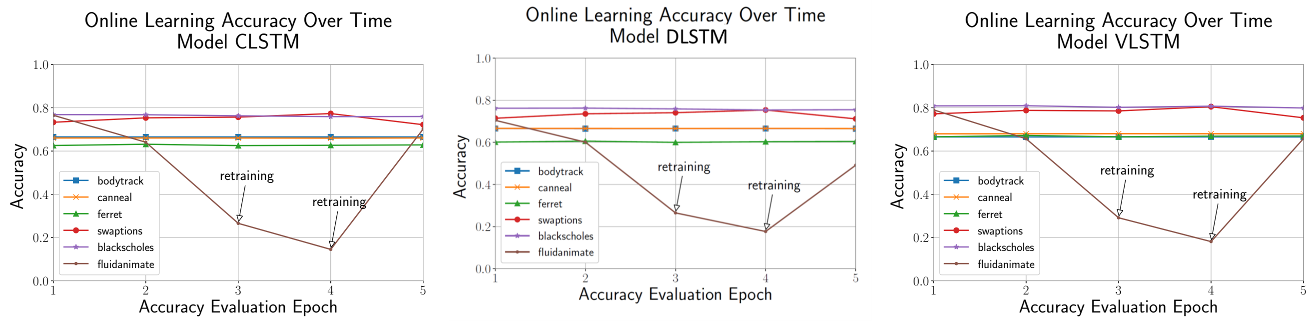

Figure 5: Online learning accuracies over time. Accuracy is reevaluated periodically and if it gets below 60%, then the model is retrained.

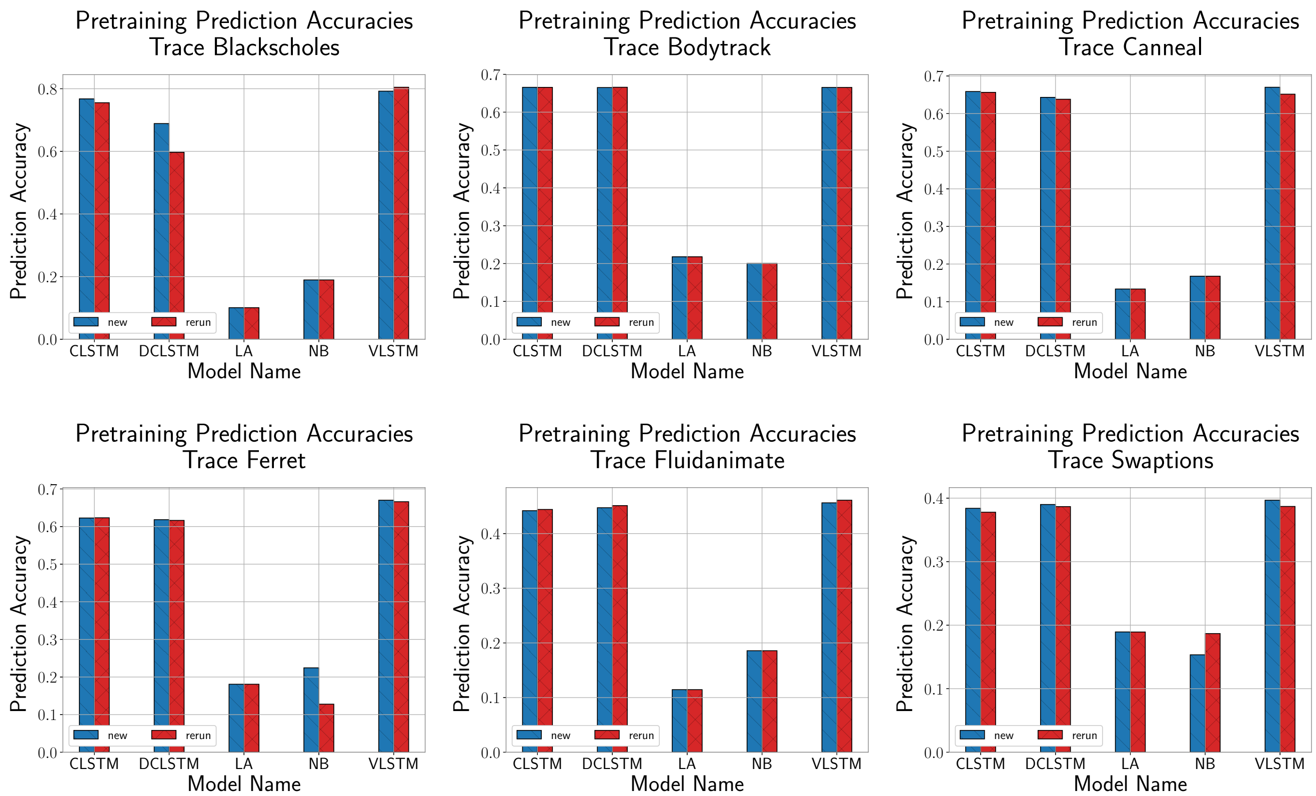

Figure 6: Prediction accuracy comparisons for each trace when tested on a rerun trace from the same app with the same input parameters, vs the same app with different input parameters.

4.6 Prefetch Evaluation

We compare a simple prefetcher based on our doubly compressed LSTM predictions against the following state-of-the-art prefetchers:

- The Best-Offset Prefetcher (BOP) [18] determines the most popular delta over a training period and maintains that delta as a consistent prediction.
- The Signature Path Prefetcher (SPP) [19] utilizes a custom signature as input - it is defined as a function of memory address delta and previous signature. During training, each signature is associated with the memory address deltas that follow it via a delta frequency dictionary. Thus, when making predictions, the frequency dictionary enables the prediction of the next delta with a certain confidence. Prediction confidence is a function of prediction decay (alpha = 0.9),

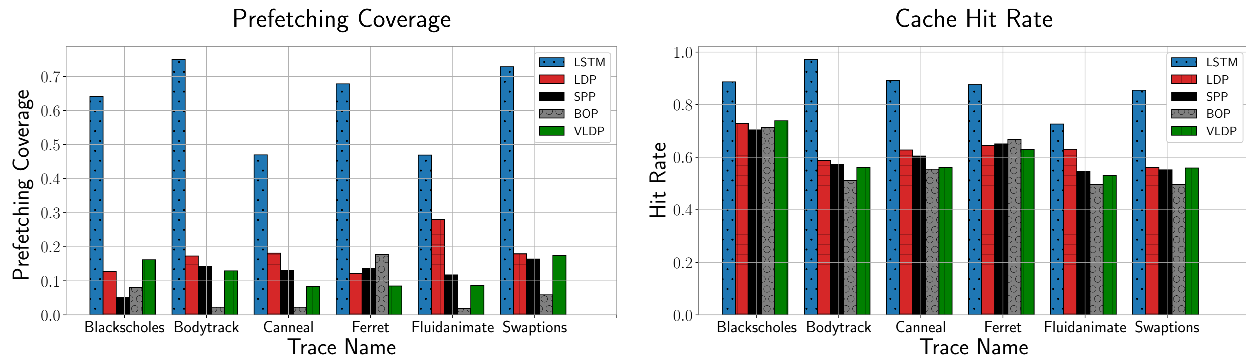

Figure 7: Prefetching coverage and cache hit rate for each prefetcher.

historical delta frequency, and prior prediction confidence. If the confidence is high enough (lower-bound = 0.3), the prefetcher transitions to the signature that follows the delta prediction and makes further predictions along a "path". This allows for predicting and fetching memory addresses several time steps into the future.

- The Variable Length Delta Prefetcher (VLDP) [20] maintains a set of dictionaries that have ordered sequences of prior address delta accesses as keys and subsequent address deltas as values. Each dictionary only stores sequences of a specific length (in our case, 1,2, and 3 – up to 2000 entries per dictionary. The model populates these dictionaries based on observed memory accesses during a training period and fetches data when past access sequences correspond to an existing key (ceding to longest observed sequence keys).
- The Last-Delta Prefetcher (LDP) is used as a basic baseline for offset prediction. For the next delta, LDP simply reuses the previous memory address delta (that occurred between the last two accesses).

Each prefetcher is used in each of the 6 traces under study. An L1 cache simulator was implemented with 64 entries, block size of 8 Bytes, and a Least Recently Used (LRU) cache replacement policy. The LSTM prefetcher uses one stream buffer that holds 8 Bytes of consecutive memory location data. In order to evaluate the prefetcher's performance, we calculate two widely used metrics, namely the cache coverage (defined as the fraction of cache misses eliminated because of prefetching), and the cache hit rate. As we can see from Fig. 7(a), the LSTM prefetcher, due to its effectiveness in modeling variable temporal dependencies, achieves significantly better coverage compared to all the other prefetchers (more than 6×) and can exceed 65% for four out of the six traces, which validates our hypothesis that LSTM can be effectively used for prefetching memory addresses. The same applies for the case of hit rate, as shown in Fig. 7(b) from where we can see that LSTM significantly increases the hit rate compared to all the other prefetchers, with the increase ranging from 10 to 40 percentage points across different traces. Note that these are full prefetchers which we are comparing against by extracting only the access prediction portion. Implementation dependent factors such as larger stream buffer sizes can significantly improve their performance results.

5 DISCUSSION

5.1 Insights

From the analysis presented in the Evaluation section, we can derive the following conclusions regarding the proposed memory prediction framework.

(1) Vanilla LSTM models provide only slightly higher accuracy compared to the compressed LSTMs at a much higher training time and memory footprint due to the large number of parameters required.
(2) Traditional models such as Naive Bayes and Logistic Regression do not provide good prediction accuracy in all the cases considered. In addition, Deep Learning models better capture the memory access prediction sequence.
(3) A relatively small number of bits (16) in the compressed prediction architecture suffices to capture majority of the variability seen in the traces.

5.2 Potential LSTM-based Prefetcher

Figure 8 provides an overview of a potential LSTM-based memory controller framework. It consists of three modules: (1) Training Module, (2) Inferencing Module, and (2) Optimization Module. The Training Module is initialized with a *Global Model* that is pre-trained based on multiple application traces. When an application starts executing, the Inferencing Module uses the Global Model to predict traces, based on which prefetching is performed. At the same time, if prediction accuracy remains lower than a pre-defined threshold (θ_1) for a certain length of the trace sequence (l_1), then the model is retrained based on the new trace sequence. Only recent traces are taken into account while training. This can be done quickly as demonstrated in our experiments. While the model is re-trained by Training Module, Inferencing Module continues to predict trace for prefetching using the previous model. This is done to ensure that **retraining does not interrupt the execution** of the application. Once the training is completed with a certain length of the trace sequence, the Inference Module switches to the *Specialized Model* for prefetching. The Specialized Model is periodically retrained as mentioned above as the behavior of the application changes and previously trained model becomes sub-optimal. The Optimization Module utilizes the predictions to make prefetch decisions, which

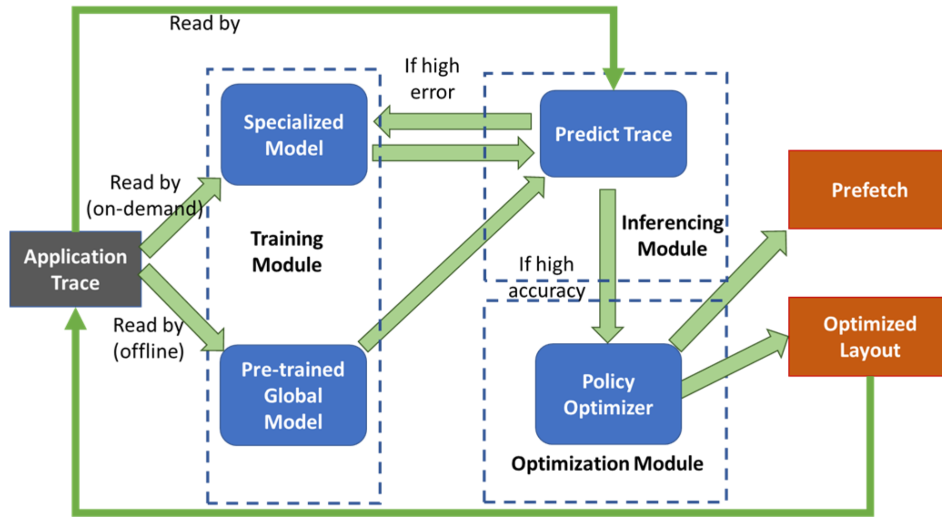

Figure 8: An overview of key components of our LSTM based prefetcher.

we will explore in future work. It considers the predicted accesses and uncertainty in predictions and addresses the following: (1) Prefetch initiation - when to prefetch, (ii) Prefetch granularity - how much to prefetch, prefetch location - where to store prefetched data (High Bandwidth Memory/cache). If the trace prediction is highly accurate for a long prediction horizon, the Optimization Module also decides if performing re-ordering of data will benefit the application, taking into account the trade-off between speedup that can be obtained due to optimized reordering and cost of re-ordering. If it identifies an advantage considering the trade-off, the application execution is interrupted and data is reorganized into the memory hierarchy.

While our prefetcher is not tied to an existing architecture, instead, it is aimed for an emerging technologies including 3D-stacked DRAM tightly integrated with FPGAs such as in Xilinx VU37P FPGA and Intel Stratix 10 MX FPGA. These technologies have enabled an architecture as shown in Figure 9, which will be the basis of our proposed prefetcher design. It consists of a hierarchy of memory: (i) far memory (e.g., such as DDR3) with high capacity, high latency, and low bandwidth, (ii) near memory (e.g., 3D-Stacked DRAM) with low capacity, low latency, and high bandwidth, and (iii) local memory (e.g., L1/L2 cache) with very low capacity, very low latency, and very high bandwidth. The data in this hierarchy is to be processed by a processor core with accelerators (Accl.) (e.g., implementing linear algebra functions for ML), and a tightly integrated embedded FPGA where our prefetcher will reside. In future work, we will explore the use of FPGAs and eFPGAs for implementing high throughput prefetching based on our compact LSTM design.

6 CONCLUSIONS

We have proposed a compression technique for LSTM that makes it a good candidate to form the basis of a prefetcher. We have shown that 100× compressed LSTM models can achieve high accuracy with small number of parameters in predicting the exact memory access when trained offline. Then, we have shown that the same

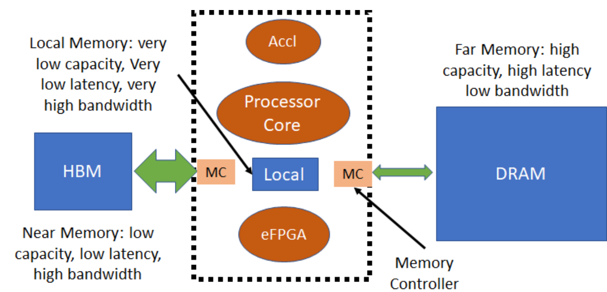

Figure 9: Target architecture.

model can also predict a rerun of the app with different model. We have further shown that these models can be quickly trained with a small dataset with small number of epochs, thus enabling online retraining. The current compression factor observed is still below the theoretically achievable $n/\log n$. In future work, we will tune the initial embedding layer to minimize the input dimensions that will bring us closer to the theoretical factor.

ACKNOWLEDGMENTS

This work is supported by Google Faculty Research Award, Air Force Research Laboratory grant number FA8750-18-S-7001, and National Science Foundation award number 1643351.

REFERENCES

[1] B. Plank, A. Søgaard, and Y. Goldberg, "Multilingual part-of-speech tagging with bidirectional long short-term memory models and auxiliary loss," *arXiv preprint arXiv:1604.05529*, 2016.
[2] O. Vinyals, Ł. Kaiser, T. Koo, S. Petrov, I. Sutskever, and G. Hinton, "Grammar as a foreign language," in *Advances in Neural Information Processing Systems*, pp. 2773–2781, 2015.
[3] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: Continual prediction with lstm," 1999.
[4] "Apple is bringing the ai revolution to your iphone." https://www.wired.com/2016/06/apple-bringing-ai-revolution-iphone/. Accessed: 2018-11-10.

[5] Y. Zeng and X. Guo, "Long short term memory based hardware prefetcher: a case study," in *Proceedings of the International Symposium on Memory Systems*, pp. 305–311, ACM, Oct. 2017.

[6] M. Hashemi, K. Swersky, J. A. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan, "Learning memory access patterns," Mar. 2018.

[7] M. F. Sakr, S. P. Levitan, D. M. Chiarulli, B. G. Horne, and C. L. Giles, "Predicting multiprocessor memory access patterns with learning models," in *ICML*, pp. 305–312, 1997.

[8] M. E. Sakr, C. L. Giles, S. P. Levitan, B. G. Horne, M. Maggini, and D. M. Chiarulli, "Online prediction of multiprocessor memory access patterns," in *Proceedings of International Conference on Neural Networks (ICNN'96)*, vol. 3, pp. 1564–1569 vol.3, June 1996.

[9] V. Fedchenko, G. Neglia, and B. Ribeiro, "Feedforward neural networks for caching: Enough or too much?," *CoRR*, vol. abs/1810.06930, 2018.

[10] F. B. Moreira, M. Diener, P. O. A. Navaux, and I. Koren, "Data mining the memory access stream to detect anomalous application behavior," in *Proceedings of the Computing Frontiers Conference*, CF'17, (New York, NY, USA), pp. 45–52, ACM, 2017.

[11] N. Zhang, K. Zheng, and M. Tao, "Using grouped linear prediction and accelerated reinforcement learning for online content caching," *CoRR*, vol. abs/1803.04675, 2018.

[12] Z. Xu, S. Ray, P. Subramanyan, and S. Malik, "Malware detection using machine learning based analysis of virtual memory access patterns," in *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '17, (3001 Leuven, Belgium, Belgium), pp. 169–174, European Design and Automation Association, 2017.

[13] L. Peled, U. Weiser, and Y. Etsion, "A neural network memory prefetcher using semantic locality," Mar. 2018.

[14] S. Rahman, M. Burtscher, Z. Zong, and A. Qasem, "Maximizing hardware prefetch effectiveness with machine learning," in *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pp. 383–389, Aug. 2015.

[15] S. Liao, T. Hung, D. Nguyen, C. Chou, C. Tu, and H. Zhou, "Machine learning-based prefetch optimization for data center applications," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pp. 1–10, Nov. 2009.

[16] A. Narayanan, S. Verma, E. Ramadan, P. Babaie, and Z.-L. Zhang, "Deepcache: A deep learning based framework for content caching," pp. 48–53, 08 2018.

[17] M. Hashemi, K. Swersky, J. A. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan, "Learning memory access patterns," *CoRR*, vol. abs/1803.02329, 2018.

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

[19] J. Kim, S. H. Pugsley, P. V. Gratz, A. Reddy, C. Wilkerson, and Z. Chishti, "Path confidence based lookahead prefetching," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, p. 60, IEEE Press, 2016.

[20] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, "Efficiently prefetching complex address patterns," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 141–152, IEEE, 2015.

[21] S. Kondguli and M. Huang, "Division of labor: A more effective approach to prefetching," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 83–95, June 2018.

[22] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, (New York, NY, USA), pp. 72–81, ACM, 2008.

[23] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, pp. 1735–1780, Nov. 1997.

[24] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. The MIT Press, 2016.

[25] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," *SIGPLAN Not.*, vol. 40, pp. 190–200, June 2005.