Long Short Term Memory Based Hardware Prefetcher

-A Case Study

Yuan Zeng
Lehigh University
yuz615@lehigh.edu

Xiaochen Guo
Lehigh University
xig515@lehigh.edu

ABSTRACT

Hardware prefetching is an efficient mechanism to hide cache miss penalties. Accuracy, coverage, and timeliness are three primary metrics in evaluating prefetcher performance. Highly accurate hardware prefetches are desired to predict complex memory access patterns in multicore systems. In this paper, we propose a long short term memory (LSTM) prefetcher—a neural network based hardware prefetcher. Offline experiment shows that the proposed LSTM prefetcher achieves higher accuracy and better coverage on a set of evaluated traces.

CCS CONCEPTS

• **Hardware → Integrated circuits**; **Semiconductor memory**; **Dynamic memory**; • **Computer systems organization → Other architectures**; **Neural networks**;

KEYWORDS

Hardware prefetcher, Neural network, LSTM, RNN, Complex access pattern, Memory hierarchy

ACM Reference format:
Yuan Zeng and Xiaochen Guo. 2017. Long Short Term Memory Based Hardware Prefetcher. In *Proceedings of MEMSYS 2017, Alexandria, USA, October 2–5, 2017,* 7 pages.
https://doi.org/10.1145/3132402.3132405

1 INTRODUCTION

The latency gap between processor and main memory continues to be a performance bottleneck. Although processor frequency does not increase anymore, the "memory wall" [24] problem is still getting worse due to the increased number of cores, and the limited off-chip bandwidth. Deep memory hierarchies help alleviate the latency problem by exploiting spatial and temporal localities, but these are not enough to solve the problem. Out-of-order execution and multithreading are useful in hiding latency in L1 caches, however, between the LLC and main memory, long memory access latency is still hindering microprocessor performance growth.

MEMSYS 2017, October 2–5, 2017, Alexandria, USA

ACM ISBN 978-1-4503-5335-9/17/10…$15.00
https://doi.org/10.1145/3132402.3132405

Hardware prefetching can captures the run time access patterns and fetch useful data before they are demanded. As a result, execution time can be improved by reduced demand cache misses or reduced miss penalties.

Accuracy, coverage, and timeliness [6] are three primary metrics in evaluating prefetcher performance. Accuracy is the number of useful prefetches divided by the number of total issued prefetches. Coverage is the number of useful prefetches divided by the total number of demand misses. Timeliness shows how close in time between the arrival of a prefetched line in cache and a demand access to that line. These three metrics have to be optimized together.

Naively increasing the total number of issued prefetches can lead to higher coverage in many cases, because increased prefetch number usually lead to more correct prediction. However, this could pollute the cache, increase the unnecessary DRAM traffic, and lead to performance degradation. So high accuracy is required. Timeliness is important too. If a useful prefetch place the data in cache too early, the prefetched line will waste precious cache space. If too late, demand access to the line will experience miss penalty. Limited off-chip bandwidth and increasing memory latency due to increased memory capacity together make inaccurate prefetching more expensive than before. Therefore, accuracy becomes more important for hardware prefetcher.

Artificial neural networks show a great potential in accurate pattern predictions. It made a big impact on image processing, audio processing, and natural language processing [25] [1] [22] . Recently, neural network have been used in computer architecture design as well and have made great contribution to system performance improvement [7] [19] [17]. Among existing neural network algorithms, *Recurrent Neural Network (RNN)*, which includes feed-back connections within the network, is especially useful in sequence predictions, and becomes a natural choice for a prefetcher.

In order to issue accurate prefetch and meet the emerging system requests, this work proposes a long short term memory (LSTM) prefetcher, which is based on a recurrent neural network algorithm-LSTM [4]. Complex memory access patterns are the main target of the proposed prefetcher, which are hard to predict and common in applications with linked data structures, compressed data formats, and data dependent control flows.

Evaluation shows that the proposed LSTM prefetcher achieves higher accuracy and coverage, as well as better noise tolerance.

2 BACKGROUND AND RELATED WORK

2.1 Prefetch Complex Patterns

Among prior prefetcher designs, memory address predictions are either based on address pattern histories, or values stored in previously accessed memory locations. Indirect memory prefetcher (IMP) [26] is an example in the later category, which can prefetch

indirect memory accesses in the form A[B[i]]. IMP specifically targets on machine learning, graph analytics, and sparse matrix based applications. The proposed prefetcher is based on pattern history, and targets on a wider range of applications that exhibit complex delta patterns.

Best Offset prefetcher (BO) [12], Access Map Pattern Matching (AMPM) [5], Spatial Memory Streaming (SMS) [21] , Global History Buffer (GHB) [14], and Variable Length Delta Prefetcher (VLDP) [20] are based on address pattern histories. BO targets on fetching data with the most frequently seen offsets. BO is good in timeliness since it uses a delay queue to record order information for each offsets. But BO is not accurate enough. Because it only fetches the most frequent patterns, and many critical but infrequent patterns can be skipped. AMPM and SMS are two designs that exploit recurring memory footprints within a spatial region. After observing access patterns within a memory region, all of the accessed blocks are marked and will be fetched together upon the next access to the same spatial region. Performance gain of this kind of prefetchers mainly comes from the high coverage. But this type of design do not consider timeliness. No information on access order is recorded within the memory region, which leads to many early prefetches. GHB and VLDP both memorize the offset differences between two adjacent accesses (*delta*) within an OS page to learn the address correlations. While GHB can only make prediction based on one delta history, VLDP uses cascaded delta prefetching table (DPT) to enable predictions based on variable delta history lengths. This kind of prefetchers can be more accurate and efficient because timeliness is considered.

Many applications have long delta sequence, which can not be effectively predicted by VLDP. In VLDP, delta sequences are recorded for each OS page in delta history buffer (DHB). When a prefetch triggering event happens (cache misses or cache hits to prefetched lines), per page delta history will be used to index to multiple global delta history tables (DPT) with different history lengths. If hit in DPT, delta associated with matching history pattern will be used to calculate the predicted address. If miss in DPT, new patterns are added to the DPT table. Existing pattern which leads to incorrect address prediction can be promoted to an upper level DPT with longer history length. In the VLDP mechanism, increasing delta history length will result in increased number of DPTs, thus leads to more storage overheads. Since every sub-sequence will be stored, the storage requirement increased exponentially. DPT table can not hold all of the history sequence within its limited number of entries. DPT miss will prevent VLDP from achieving higher coverage. Incomplete history will limit the accuracy. The proposed design aims at finding a new solution to the problem of predicting long history sequence accurately with high coverage.

The proposed design is an improvement on the VLDP, some similarities include: 1) LSTM prefetcher is recording successive delta history within OS page boundary, and making predictions based on global delta patterns. 2) LSTM prefetcher takes action on the same prefetch trigger events as the VLDP does. However, LSTM prefetcher uses neuron network algorithm, rather than the cascaded tables to predict long delta sequence. LSTM prefetcher can predict based on learned pattern all the time, rather than VLDP's "reactive" way (only update global history when miss prediction occurs). These improvement can help LSTM prefetcher to achieve higher accuracy and coverage as compared to VLDP.

Signature Path Prefetcher (SPP) [9] improves VLDP design as well. However, it gains performance improvement mainly by increasing the prefetcher depth, which leads to more aggressive prefetching. LSTM prefetcher can be more aggressive by issuing multiple prefetches upon one demand access, but the main performance gain comes from accuracy and coverage improvement. SPP stores delta history within an OS page boundary too, but in a compressed fashion. SPP is still a table based design with limited number of table entries, it can not store all of the useful history information. The compressed history can lead to more aliasing, which prevents SPP design from accurate predictions.

2.2 Machine Learning in Prefetcher Design

There are several prior prefetchers that uses machine learning algorithms. One type of designs use machine learning to select the best performed prefetcher [18] or optimal prefetcher configuration [10]. Another uses machine learning to detect patterns. LSTM prefetcher falls into the second category. A prior work named adaptive prefetching (AP) [3] uses neural network to predict patterns. This design uses a table to record memory reference and simplify the information needed for training. The input of the network is table indices. AP records memory access history in a table, which prevents it from capturing all of the useful information. AP design uses a single layer time delay neural network (TDNN), while LSTM prefetcher is based on RNN, which is a more accurate algorithm for time series prediction.

2.3 LSTM Algorithm

LSTM [4] is one of the RNN algorithms, which is especially useful in long term dependence sequence prediction. Because LSTM solves the vanishing gradient problem [2] in traditional RNN. Fig. 1 shows a simple LSTM network used to predict a sequence of odd numbers. The given network has three layers: input layer, output layer, and a single hidden layer. Only one LSTM block exists in the hidden layer. Arrows among the input layer, the output layer, and the LSTM blocks represent parameters in the network, which includes *weights* and *bias*. Weights show how strong each connection is, and bias are extra values added to each connection, which are used to shift the input-output curve in order to generate accurate predictions.

Like other machine learning algorithms, LSTM performs two operations: prediction and training. During the prediction process, inputs are given one by one to the network in sequence. When one system input is given to the network (e.g., 3 in Fig. 1 at T2), one output will be computed (4 in Fig. 1 at T2), which is the prediction result. Then the network parameters are updated to minimize the difference between the predicted result (4 in T2) and the actual next input (5 at T3) (*loss*), which is the training process. We refer to the interval between an input given to the network and the network parameters fully updated as a *time step* (e.g, in Fig. 1, T1, T2, T3 are three time steps).

Each output is used as part of the block input in the next time step. The block input comprises of two parts, current system input (one data within the sequence), and the output of the network generated by a previous input. An unfolded structure of LSTM network is

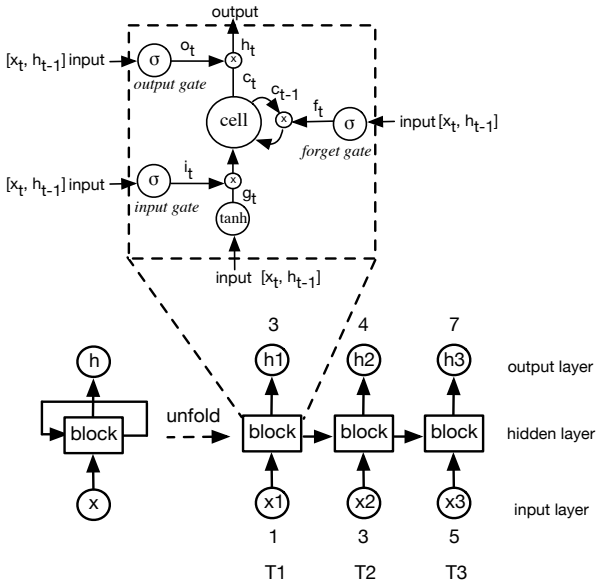

Figure 1: Illustration of the LSTM Algorithm.

shown in Fig. 1 to demonstrate inputs and outputs at different time steps. After the network is iteratively trained, the predicted result is expected to match incoming data for sequences that have patters.

Each LSTM block stores a cell state, which integrate information from different *gates* that controls the amount of information add to or remove from the cell. The gate is composed by a sigmoid layer and a dot product operation. Three gates exist within each block, an input gate, an output gate, and a forget gate. Since sigmoid function outputs between 0 and 1, these gates can let entire information flow through by outputting 1, or let nothing flow through by outputting 0. The gate design is the key for the LSTM algorithm, which gives LSTM network the ability to remember long-term dependencies. A tanh layer creates a vector of new candidate input values to the cell.

The prediction process is also called “front propagation (FP)", and the training process is referred to as “back propagation (BP)". Eq 1 to Eq. 6 shows the prediction process at time step t. x_t is one of the data in a given sequence. h_{t-1} is the predicted output at previous time step. i_t, f_t, o_t are three gate control values between 0 and 1. g_t is the currently generated input vector to the cell. c_{t-1} is the previous cell state. c_t is current cell state. h_t is the final output of the network, which is the predicted data.

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{1}$$

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{2}$$

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{3}$$

$$g_t = tanh(W_g \cdot [h_{t-1}, x_t] + b_g \tag{4}$$

$$c_t = f_t \times c_{t-1} + i_t \times g_t \tag{5}$$

$$h_t = o_t \times c_t \tag{6}$$

In the back propagation process, total system loss from time 1 to time T is given by $L_{(t)}$ in Eq. 7. New weights and bias need to be calculated for each training process to minimize $L_{(t)}$. *Gradient descent* algorithm is typically in the BP process. In gradient descent algorithm, partial derivatives of all of the weights and bias, including $W_i, W_f, W_o W_g, b_i, b_f, b_o, b_g$, are calculated by the derivative chain rule. In the equation, T is the total number of time steps. Leaning rate α represents the learning speed, the parameters are updated by subtracting the product of learning rate and parameter derivatives (Eq. 8 and Eq. 9). Bias are calculated and updated in a similar manner.

$$L_{(t)} = \sum_{t=1}^{T} (h_t - y_t)^2 \tag{7}$$

$$w_o -= \frac{dL}{dw_o} \times \alpha, w_i -= \frac{dL}{dw_i} \times \alpha \tag{8}$$

$$w_f -= \frac{dL}{dw_f} \times \alpha, w_g -= \frac{dL}{dw_o} \times \alpha \tag{9}$$

3 LSTM PREFETCHER ARCHITECTURE

The proposed prefetcher is attached to the last level cache (LLC) and snoops every LLC demand hit and miss (Fig. 2). This prefetcher relies on local delta history to predict the addresses of future memory accesses.

The LSTM prefetcher predicts within the OS page boundary. It uses a local history table (Fig. 2(1)) to record the local history information; an offset table (Fig. 2(2)) to hold the first offset-delta pair within a page; and an LSTM hardware data path train parameters and predict addresses based on the history information. Every demand access first checks the table to get history information before starting the prediction process. The entire prediction process includes two steps: delta prediction and system training. After the prediction process, local history table will be updated. In Fig. 2, hardware components (1), (2), (3), (4), (5), (6), (7) will be used for delta prediction, and (7), (8), (6), (4), (3) will be used for system training.

3.1 Meaningful LLC Accesses

The Local history table holds the the page numbers and recent page histories for different OS pages. Each local history table access searches all of the local history table entries to check if there is a page number match. Each entry has four data fields: 1) page offset of the previously accessed address (preAddr) 2) four lasted delta values (pre4Delta), 3) four page offsets of the recently issued prefetches (pre4FetchAddr), 4) state information for the local delta sequence (preState). We choose to store four previous deltas because experiments show that the history length of four can provide enough information for accurate predictions.

Although LSTM prefetcher checks every LLC access, it only activates the prediction process on meaningful LLC accesses, which captures the LLC demand misses as if no prefetch operation ever happened. A meaningful LLC access is either a demand miss or a demand hit on a previously prefetched cache block. When a new

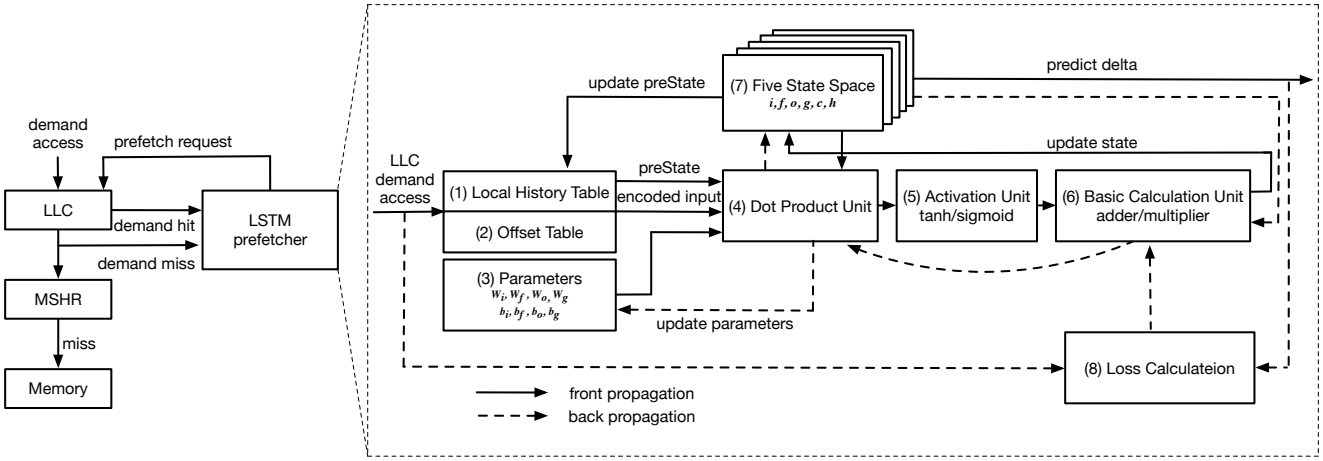

Figure 2: LSTM Prefetcher Architecture.

access enters the LSTM prefetcher, page number is used to search the page number field of the local history table. If the access misses in the local history table, it is a meaningful access and a new entry will be added. The least recently used entry will be evicted if the local history table is full. If the access hits in the local history table and the offset matches one of the pre4FetchAddr, this access will be considered as a meaningful access, which will activate a prefetch.

3.2 Prediction Processes

The offset table is indexed by the page offset of the first access to a page, and each entry stores a delta value associated with the page offset. On the first meaningful LLC access to a page, offset of the address will be used to search the offset table. If no match, a new entry will be allocated. On the second access to the same page, delta is calculated and stored in the offset table associated with the offset of the first access. If on the next hit to the offset table, the delta stored with the offset will be used to calculate the prefetch address.

After the second meaningful LLC access to a page, local history table will be searched. If there is a local history table hit, a delta value will be calculated using the current offset and the offset of the previous access to this page. The four history deltas, current calculated delta, and the local history table index are represented in binary format and input to the LSTM. The reason of having both the history delta and the local history table index as inputs is to provide opportunities for parameter sharing and to help reducing the aliasing problems. Because patterns from different pages are sharing the same set of LSTM parameters, using local history table index as input can provide additional information to distinguish from different pages, and to establish correlations within the same page. In the prediction process, parameter sharing explores inter-page correlations, which can lead to more prediction opportunities.

Inputs are given to the network in different time steps one by one in sequence. The proposed design includes five time steps because five delta histories are used. At each time step, 6 state value vectors $(i_t, f_t, o_t, g_t, c_t, h_t)$ for each LSTM block are calculated (Eq. (2.1) to Eq. (2.6)) and stored in one of the state space (Fig. 2(7)). Among them the output vector (h) and current cell state vector (c) will be used for the state value vectors calculation in the next time step. After the 5th state value vectors are calculated, the output vector without the local history page table index will be used as the predicted delta value. Since the oldest delta history in local history table will be evicted when new delta comes, the output vector (h) and current cell state vector (c) stored at the first state space will be record to the local history table in order to initial next delta prediction within the same page.

Upon observing the next access to the same page and calculating the new delta, the difference between the correct output and the last predicted output (h) is calculated by Eq. (2.7). In order to minimize loss, partial derivatives of the parameter matrices need to be computed and used for updating parameters. This is the system training process. After the LSTM network gets enough amount of training with an appropriate learning rate, the parameters will converge, and the correct prediction can be expected when repeating patterns occur.

3.3 Hardware Implementations

In order to implement the LSTM algorithm introduced in Section 2.3, an LSTM data path is built in hardware, which includes a table storage, a run-time state storage and functional units. Although local history for each OS page is stored separately in local history table, only one LSTM data path is required to predict addresses based on both local and global histories. All pages share the same set of LSTM parameters (Fig. 2(3)), which are initiated to randomly generated values between -10^{-5} and 10^{-5}.

The five state space is the runtime storage space to store all of the state value vectors which have been calculated during the five time steps. Those vectors need to be stored because they will be used to update parameters sequentially after the delta prediction process. However, unlike parameters which will be held during the entire application running process, at the end of each prediction process, the state space will be flushed and all of the internal values will be discarded after the parameter updates.

In order to calculate the state vectors and to update the parameters, serval functional units are required. Dot product unit (Fig. 2(4)) is used for matrix multiplication. The activation unit (Fig. 2(5)) applies tanh and sigmod activation functions to calculate the values

for state vector i, f, o, g. The basic calculation unit (Fig. 2(6)) includes adders and multipliers, which help satisfy the rest of the computational requirements. The loss calculation unit (Fig. 2(8)) is part of the basic calculation unit, which is shown separately in Fig. 2 because the loss function is only calculated for the training process.

Hardware implementation of neural network algorithm has been a challenge, since it requires a large amount of computation and storage. However, by using emerging technology, LSTM algorithm can be implemented with affordable hardware overhead and acceptable latency. Dot product unit can be built using emerging resistive crossbar array [8], which maps the input vector and weight matrix to the input voltage and inductance of a resistive crossbar array. Matrix-vector multiplication can be done by simply sampling the current flowing in each row of the resistive crossbar array. By using this technology, dot product can be done within a single clock cycle. The activation unit can be implemented by look up table (LUT) [13] [16] [15] [23], in which the function curve is divided into different segments and the corresponding output values for each input segment are stored in a table. Only one memory access time is required to get the function outputs, and the storage overhead is proposed to be only 60B in a recent sigmoid function implementation [11], tanh function can be implemented in the same way as well.

4 CASE STUDY

To analyze the advantages and disadvantages of the proposed LSTM prefetcher, we conduct an offline case study on selected trace.

4.1 Prefetcher Configurations

We simulated the proposed LSTM network with a 64-entry offset table, a 64-entry local history table, and an LSTM network with realistic configurations. 4KB OS page size and 64B cache block size are used. Page offset ranges from 0 to 63, and the delta value is between -63 to 63. We use a 7-bit 2's complement binary representation for deltas, and a 6-bit binary representation for the local history table index. The total input length is 13 bits.

In the LSTM network, 13 LSTM cells form a hidden layer, which connect to 13 input and 13 output nodes. The network structure is shown in Fig. 3. There are four weight matrices (W_g, W_i, W_f, W_o) related to each gate within each cell, each has a size of 13*26 words. Each gate also requires four bias vectors (b_g, b_i, b_f, b_o), each of which has 13 words. We choose history length of five, thus, five history deltas are used as the inputs in five time steps to make the prediction. Each of the five state spaces contains six state vectors (i, f, o, c, s, h) with a size of 13 word. Data stored in the matrices are all 32 bit floating point values. The learning rate various for different traces in order to get best performance result for the system. Storage overhead are shown in Table 1. Using fixed-point representation is a common practice to reduce hardware complexity. According to our analysis on the dynamic range of the data, 16 bit fixed point representation can provide enough accuracy. If we use the 16-bit fixed-point data representation, the total storage overhead for this design will be 7.4KB.

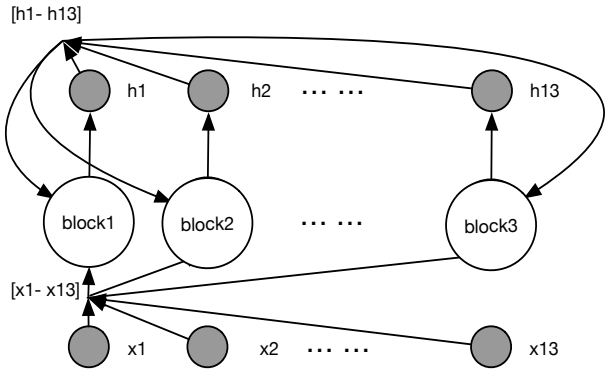

Figure 3: Network structure.

Table 1: LSTM Prefetcher Storage Overheads.

Component	Size in Bit	Size in Byte
Offset Table	$64 \times (6 + 7)$	104B
Local History Table	$64 \times (20 + 28 + 24 + 832)$	7232B
Weight Matrices	$4 \times 13 \times 26 \times 32$	5408B
Bias Vectors	$4 \times 13 \times 32$	208B
State Vectors	$5 \times 6 \times 13 \times 32$	1560B
Total		14.17KB

4.2 Offline Results and Analysis

Both LSTM prefetcher and VLDP are implemented offline without multi-degree prefetching. Generated traces are used to test the accuracies and coverages of these two prefetchers. The traces are represented by regular expressions. To ensure fairness, the implemented VLDP is built with five DPTs to support delta pattern searches with history length of five, which is the same as the number of time steps in the LSTM prefetcher. Storage Overhead of the three-DPT implementation reported in VLDP paper is 998B. The five-DPT VLDP implementation exhibits a 1.37KB storage overhead. The number of useful prefetches is counted by comparing each newly arrived delta value with the last prefetched delta value. Accuracy results and the number of issued prefetches are shown in Table 2.

VLDP is constraint by the limited storage space. Because it can only hold limited number of cascaded DPTs, and each table has finite number of entries. However, this problem does not exists in LSTM prefetcher, because the LSTM network remembers the entire sequence directly. Trace 1-3 shows the repeating sequences of 65, 75, 85 different deltas which are randomly generated. VLDP can not make any prediction because the useful sub-string information will be evicted before it is encountered again. For example, when (a_1, a_2) is accessed, sub-string (a_1, a_2) will be recorded to the DPT. In the following accesses, (a_2, a_3), (a_3, a_4) ... (a_{64}, a_{65}) will be recorded in the DPT table. Since the DPT table has limited number of entries, (a_1, a_2) will be evicted before the next (a_1, a_2) pattern is encounted. As a result, VLDP will lose all of the prediction opportunities. The result shows that VLDP issues zero prefetch, and the accuracy is zero. However, LSTM prefetcher has more potential in predicting long sequence. The test results show that LSTM can achieve 61.1%

Table 2: Comparison between VLDP and LSTMP Prediction Accuracy.

Trace	Accuracy for VLDP\LSTMP	Num of Issued Prefetch by VLDP\LSTMP
1. $a_1a_2a_3...a_{65}$	0% \ 61.1%	0 \ 31959
2. $a_1a_2a_3...a_{75}$	0% \ 44.6%	0 \ 37532
3. $a_1a_2a_3...a_{85}$	0% \ 38.9%	0 \ 42770
4. Random noise trace A	74.3% \ 85.6%	17492 \ 17482
5. Random noise trace B	62.9% \ 82.8%	17493 \ 17488
6. Random noise trace C	71.5% \ 86.8%	17495 \ 17296

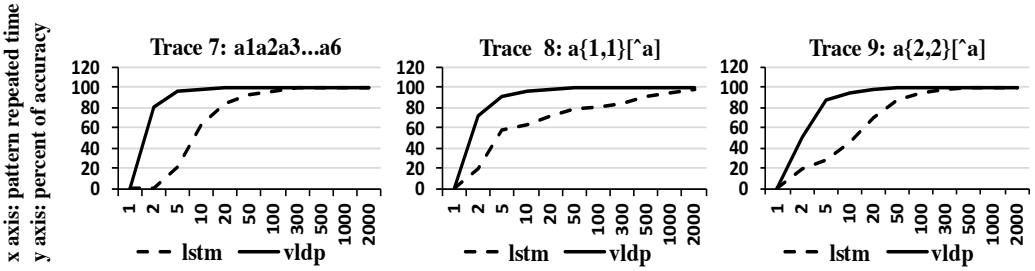

Figure 4: Warm-up time comparison between VLDP and LSTMP.

accuracy for delta length 65, 44.6% accuracy for delta length 75, and 38.9% accuracy for delta length 85.

Using previous state of an newly evicted delta as one of the prediction input is another specific design choice made for the proposed LSTM prefetcher. Since local delta history shift left on each access, certain delta will be evicted after four following accesses in the same page, all of the input data will be trained for five times. When a delta is evicted, it can not be trained in future accesses. However, the information of the evicted delta is still used in future predictions. Therefore, the actual local history LSTM prefetcher is more than five. This property can help increase accuracy because longer history is considered.

LSTM prefetcher is also good at tolerating noise. Trace 4-6 includes repeating delta (a) with randomly generated noise inserted. LSTM prefetcher will not be influenced by the noise and continuously predicting (a) correctly, while for VLDP, the noise will be recorded and lead to mis-prediction. Three tests shows that LSTM prefetcher can achieve 85.6%, 82.8%, and 86.8% accuracy. VLDP can achieve 74.3%, 62.9% and 71.5% accuracy respectively.

4.3 Challenges and Potential Solutions

Despite high prediction accuracy and good noise tolerance, LSTM prefetcher could potentially suffer from long prediction latency, large storage overheads, and slow warm up. When making predictions, table based schemes use local delta string to search the global table, while LSTM prefetcher needs to calculate the state vectors. Calculation takes longer time than searching the tables, which can be a potential issue for integrating LSTM prefetcher into the system. One possible solution to alleviate long prediction latency is to combine a simple next-line prefetcher, and only use LSTM prefetcher to predict complex patterns because LSTM prefetcher is inefficient when predicting simple trace. When good spatial locality is detected, next-line prefetcher can be used to accelerate the prediction.

The storage overhead is a potential disadvantage for LSTM, although 14.17KB storage overhead for LSTM is smaller than the 22KB storage overheads in SMS [21], it is much larger than the storage overheads of VLDP. Using fixed-point inner data representation can decrease the storage overhead for half. Using other RNN algorithm like GRU, which has similar functionality as LSTM but has less parameters, will also help to reduce the overhead.

Warm-up time is another potential drawback for LSTM prefetcher. In the beginning of the entire access string, LSTM needs to get four to five inputs before it can make a prediction, because the LSTM network needs time to get trained. Figure 4 shows the warm-up time comparison between VLDP and LSTM prefetcher. For different traces, VLDP perform better at the beginning, but LSTM prefetcher ends up with the similar accuracy and coverage as VLDP after the warm up. This warm up problem can be alleviated by the global pattern sharing mechanism. If pattern (a, b, c) is a pattern which has already appeared for serval times globally, when a new page is accessed, even if delta (a) is observed in the first access, a prediction (b) can still be made. However, in some cases where repeating pattens exists in the beginning of the access sequence within a single page, LSTM prefetcher will not perform well.

5 CONCLUSION

This work proposes a LSTM prefetcher to accurately predict complex memory access patterns. Results for 9 traces show that LSTM prefetcher is good at remembering long access sequence while traditional prefetcher like VLDP suffers from storage problem. LSTM can tolerate random noise, whereas patterns recorded by VLDP can be disturbed. LSTM prefetcher suffers from long warm-up time and prediction latency, which make it inefficient when predicting the simple trace. LSTM prefetcher also requires relatively larger storage overheads, which needs to be improved.

6 ACKNOWLEDGMENTS

This material is based upon work supported by a startup grant from Lehigh University. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of Lehigh University. The authors would like to thank anonymous reviewers for helpful feedback.

REFERENCES

[1] Dario Amodei, Rishita Anubhai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, et al. 2015. Deep speech 2: End-to-end speech recognition in english and mandarin. *arXiv preprint arXiv:1512.02595* (2015).

[2] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. 1994. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks* 5, 2 (1994), 157–166.

[3] John Cavazos and Darko Stefanovic. 1997. Adaptive Prefetching using Neural Networks. *Proposal to NEC*.

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

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

[6] Hyesoon Kim Jaekyu Lee and Richard Vuduc. 2012. When prefetching works, when it doesn?t, and why. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 1 (2012), 2.

[7] Daniel A Jiménez and Calvin Lin. 2001. Dynamic branch prediction with perceptrons. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*. IEEE, 197–206.

[8] Hyungjun Kim, Taesu Kim, Jinseok Kim, and Jae-Joon Kim. 2017. Deep Neural Network Optimized to Resistive Memory with Nonlinear Current-Voltage Characteristics. *arXiv preprint arXiv:1703.10642* (2017).

[9] Jinchun Kim, Seth H Pugsley, Paul V Gratz, AL Narasimha Reddy, Chris Wilkerson, and Zeshan Chishti. 2016. Path confidence based lookahead prefetching. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 1–12.

[10] Shih-wei Liao, Tzu-Han Hung, Donald Nguyen, Chinyen Chou, Chiaheng Tu, and Hucheng Zhou. 2009. Machine learning-based prefetch optimization for data center applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 56.

[11] Pramod Kumar Meher. 2010. An optimized lookup-table for the evaluation of sigmoid function for artificial neural networks. In *VLSI System on Chip Conference (VLSI-SoC), 2010 18th IEEE/IFIP*. IEEE, 91–95.

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

[13] A Muthuramalingam, S Himavathi, and E Srinivasan. 2008. Neural network implementation using FPGA: issues and application. *International journal of information technology* 4, 2 (2008), 86–92.

[14] Kyle J Nesbit and James E Smith. 2004. Data cache prefetching using a global history buffer. In *Software, IEE Proceedings-*. IEEE, 96–96.

[15] Amos R Omondi and Jagath C Rajapakse. 2002. Neural networks in FPGAs. In *Neural Information Processing, 2002. ICONIP'02. Proceedings of the 9th International Conference on*, Vol. 2. IEEE, 954–959.

[16] F Piazza, A Uncini, and M Zenobi. 1993. Neural networks with digital LUT activation functions. In *Neural Networks, 1993. IJCNN'93-Nagoya. Proceedings of 1993 International Joint Conference on*, Vol. 2. IEEE, 1401–1404.

[17] Demetri Psaltis, Athanasios Sideris, and Alan A Yamamura. 1988. A multilayered neural network controller. *IEEE control systems magazine* 8, 2 (1988), 17–21.

[18] Saami Rahman, Martin Burtscher, Ziliang Zong, and Apan Qasem. 2015. Maximizing hardware prefetch effectiveness with machine learning. In *High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on*. IEEE, 383–389.

[19] Sam Romano and Hala ElAarag. 2011. A neural network proxy cache replacement strategy and its implementation in the Squid proxy server. *Neural computing and Applications* 20, 1 (2011), 59–78.

[20] Manjunath Shevgoor, Sahil Koladiya, Rajeev Balasubramonian, Chris Wilkerson, Seth H Pugsley, and Zeshan Chishti. 2015. Efficiently prefetching complex address patterns. *Proceedings of the 48th International Symposium on Microarchitecture*, 141–152.

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

[22] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. 2012. LSTM Neural Networks for Language Modeling.. In *Interspeech*. 194–197.

[23] Ngah Syahrulanuar, Abu Bakar Rohani, and Embong Abdullah. 2014. Two-Step Implementation of Sigmoid Function for Artificial Neural Network in Field Programmable Gate Array. (2014).

[24] Wm A Wulf and Sally A McKee. 1995. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news* 23, 1 (1995), 20–24.

[25] Zichao Yang, Xiaodong He, Jianfeng Gao, Li Deng, and Alex Smola. 2016. Stacked attention networks for image question answering. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 21–29.

[26] Xiangyao Yu, Christopher J Hughes, Nadathur Satish, and Srinivas Devadas. 2015. IMP: Indirect memory prefetcher. In *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 178–190.