2017

Long Short Term Based Memory Hardware Prefetcher

Yuan Zeng
Lehigh University

Long Short Term Memory Based Hardware Prefetcher

By

Yuan Zeng

A Thesis

Presented to the Graduate and Research Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Electrical Engineering

Lehigh University

May, 2017

This thesis is accepted and approved in partial fulfillment of the requirements for the Master of Science.

Date

Xiaochen Guo
Thesis Advisor

Svetlana Tatic-Lucic
Chairperson of Department

Contents

List of Figures

List of Tables

Abstract

Hardware prefetching is an efficient way to hide cache miss penalty due to long memory access latency. Accuracy, coverage, and timeliness are three primary metrics in evaluating hardware prefetcher design. Highly accurate hardware prefetches are required 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 to achieve high prefetch accuracy and coverage while improving prefetch timeliness. The proposed LSTM prefetcher achieves higher accuracy and coverage by training neural networks to predict long memory access patterns. LSTM can improve timeliness in two approaches. First, multiple prefetch can be issued on a single cache access. Second, a simple Next-N-Line prefetcher is integrated with the LSTM prefetcher to accelerate predictions when good spatial locality exists. The proposed LSTM prefetcher is the first prefetcher design that uses recurrent neuron network. Three case studies are presented, which show that proposed LSTM prefetcher can achieve 98.6%, 83.5%, and 61% accuracy respectively, while the state-of-art variable length delta prefetcher (VLDP) achieves 0%, 75% ,and 26.6% accuracy in predicting the sequences in the case studies.

Chapter 1

Introduction and Motivation

The latency gap between processor and main memory continues to be a bottleneck. Although processor frequency no longer grows, the "memory wall" [1] problem is still getting worse due to the increased number of cores, and the limited off-chip bandwidth in the multicore era. Complexed 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 the show-stopper that hinders microprocessor performance scaling.

Hardware prefetching is a powerful and well-studied approach in dealing with long memory access latency. It can captures the run time access patterns and fetch useful data earlier before they are demanded. As a result, performance can be improved due to decreased demand cache misses or reduced miss penalty.

Accuracy, coverage and timeliness [2] are three primary metrics in evaluating prefetcher efficiency. Accuracy is the number of useful prefetches divide by the number of total issued prefetches. Coverage is the number of useful prefetches divide by the total number of demand misses. Timeliness shows how long it is before a prefetched cache line is touched by a demand access. These three metrics have to be optimized together.

High coverage can be achieved by naively increasing the total number of issued prefetches. 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 precise cache space. If too late, demand access to the line will experience miss penalty.

Among prior prefetcher designs, next-N-line prefetch [3] is simple and effective. Later improvements of this design include using a stream buffer to track multiple interleaved streams in parallel while avoiding cache pollution (stream prefetcher) [4] , predicting stride rather than always fetch the adjacent data (stride prefetcher) [5], and using feedback mechanisms to control aggressiveness (Feedback Directed Prefetcher [6]). These designs work well when good spatial locality exists. However, complex patterns exhibit applications with linked data structure, compressed data format, and data dependent control flow. In order to keep high accuracy while achieving good coverage for these applications, advanced prefetcher are proposed.

Sandbox prefetcher (SBP) [7] design uses a bloom filter to find the best performed prefetcher among several aggressive prefetheres. Best Offset Prefetcher (BO) [8] targets on fetching data with the most frequently seen offset after learning memory access patterns. BO is good in timeliness since it uses a delay queue to record order information for each offsets. But both SBP an BO are not accurate enough. Because they only fetch the most possible patterns, many critical patterns that do not appear frequently can be skipped.

Access map pattern matching (AMPM) [9] and Spatial memory streaming (SMS) [10] are two designs that exploit recurring memory footprints within a spatial region to learn complex patterns. 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. By prefetching all of the spatially correlated data within a region, more prefetching chance can be exploited. Although these prefetchers are more accurate than previous ones in fetching complex patterns, they do not consider the timeliness issue. No information on access order is recorded within the memory region. Thus lead to many early prefetches, which prevent AMPM and SMS from effectively utilizing caches and bandwidth.

Global History Buffer (GHB) [11] and Variable Length Delta Prefetcher (VLDP) [12] both memorize the offset difference between two adjacent accesses (delta) within an OS page in the history order to learn the data correlation. While GHB can only make prediction based on one delta history, VLDP uses cascaded delta prefetching table (DPT) to enable prediction on multiple delta history length. This kind of prefetchers can be more accurate and efficient because timeliness is considered. However, many applications have long delta sequence while VLDP is limited to recognizing short delta patterns. In VLDP, increasing in recorded delta length will result in adding additional DPT tables, thus lead to more storage overheads. Since every sub-sequence will be stored when recording a string of delta, the storage requirement increased exponentially. DPT table can not hold all of the history sequence within its limited entry. DPT miss will prevent VLDP from achieving higher coverage. The proposed design aims at finding a new way to solve the problem of recording long history delta sequence with high coverage.

Artificial neural networks show a great potential in accurate pattern prediction. It has a huge impact in image processing, audio processing, and natural language processing [13] [14] [15] . Recently, neural network have been widely used in computer architecture design and have made great contribution in improving the system performance [16] [17] [18]. Among existing neural network algorithms, Recurrent Neural Network (RNN), which includes feed-back connections within the network, is especially powerful in sequence predicting, and becomes a natural choice to predict memory access sequence.

Nowadays, bandwidth per core becomes a major limitation in multi-core system, and larger DARM capacity lead to even longer access latency, which together make mis-prefetch more expensive than before. In this case, accuracy becomes more and more important. In order to capture the missing chance for previous VLDP design (which is the best approach in balancing accuracy, coverage and timeliness in complex pattern predicting area) and meet the emerging system requests, we proposed the long short term memory prefetcher (LSTMP), which is based on a powerful neural network algorithm [19]. The contributions of this work include:

1) This is the first prefetcher design that uses recurrent neuron network (RNN)

2) This design achieves higher accuracy and coverage by the ability of recording long history delta sequence using LSTM network parameters

Chapter 2

Background

2.1 Prefetch Complex Pattern

Among prior prefetcher designs, memory address predictions are either based on memory address pattern histories, or values stored in previously accessed memory locations. Indirect memory prefetcher (IMP) [20] is an example in the later category, which can prefetch indirect memory accesses in the form A[B[i]]. IMP specifically targeting on machine learning, graph analytics, and sparse matrix based applications. The proposed design predict memory addresses based on delta history, and targeting on a wider range of applications which include complex delta patterns.

Variable length delta prefetcher [12] is one of the prior work on complex memory access pattern prediction based on history address sequence. In this design, memory access delta sequence are recorded for each OS page in delta history buffer (DHB). When prefetch triggering event happens, which include cache misses and 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.

Our design is an improvement on VLDP, some similarity includes: 1) LSTMP is recording successive delta history within OS page boundary, and making prediction based on global delta patterns. 2) LSTMP takes action on the same prefetch trigger event as VLDP does. However LSTMP uses neuron network algorithm, rather than the cascaded tables to predict long delta sequence. 2) LSTMP 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 LSTMP to achieve higher accuracy and coverage as compared to VLDP.

Signature Path Prefetcher (SPP) [21] improves VLDP design too. However, it gains performance improvement mainly by increasing the prefetcher depth, which lead to more timely prefetching. LSTMP can be more timely by issuing multiple prefetches upon one demand access, but the main performance gain comes from accuracy improvement. SPP stores delta history within an OS page boundary too, but in a compressed way. However, SPP is still a table based design with limited number of table entries, it can not store all of the useful history information. And the compressed history can lead to more aliasing, which prevent SPP design from accurate prediction.

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 [22] or optimal prefetcher configuration [23]. Another uses machine learning to detect patterns. LSTMP falls into the second category. A prior work named adaptive prefetching (AP) [24] also uses neural network to predict patterns. This design uses a table to record memory reference and simplify the information needed for training. The input for the network is table indices. AP record memory access history in table, which prevent it from capturing all the useful information. AP design uses a single layer time delay neural networks (TDNN), while LSTMP is based on RNN, which is a more accurate algorithm for time series prediction.

2.3 LSTM Algorithm

LSTM [19] is one of the RNN algorithms, which is especially useful in long term dependence sequence prediction. Because LSTM solves the vanishing gradients problem [25] in traditional RNN. Fig. 2.1 shows a simple LSTM network used to predict odd sequence. The given network only have 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 correct results.

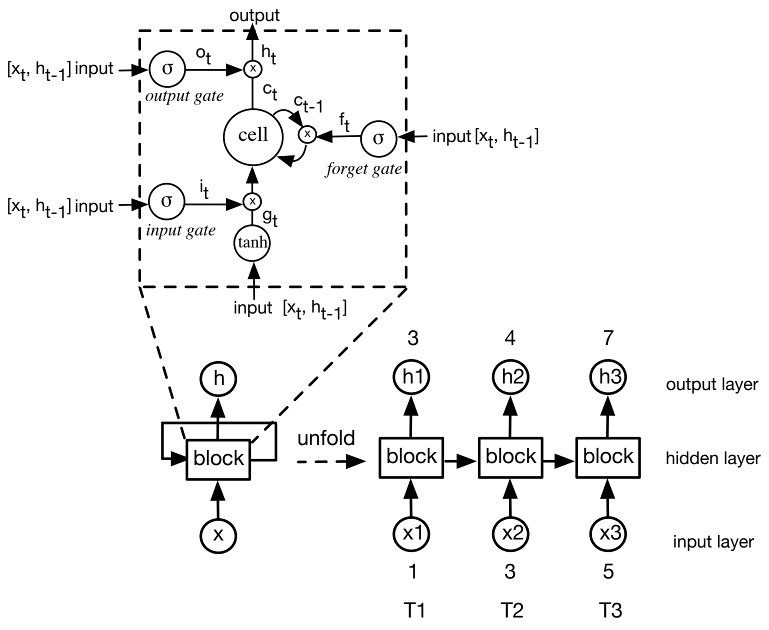

FIGURE 2.1: Long short term memory basic

Like other machine learning algorithms, LSTM algorithm has two operations: training and predict. During the prediction process, data is given one by one to the network in sequence. When one system input is given to the network, for example 3 in Fig. 2.1 at T2, one output will be computed (4 in Fig. 2.1 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. 2.1, 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 part, 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 shown in Fig. 2.1 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.

A single cell exists within each LSTM block. The cell state is like a conveyor belt which let information flow through. A tanh layer creates a vector of new candidate input values to the cell. A structure called gate controls the amount of information add to or remove from the cell, it is composed by a sigmoid layer and a point multiplication 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.

The prediction process is also called "front propagation (FP)", and the training process is referred to as "back propagation (BP)". Eq. (2.1) to Eq. (2.6) shows the prediction process at time step t. x_t is one of the data in the 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.

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{2.1}$$

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{2.2}$$

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{2.3}$$

$$g_t = tanh(W_g \cdot [h_{t-1}, x_t] + b_g) \tag{2.4}$$

$$c_t = f_t \times c_{t-1} + i_t \times g_t \tag{2.5}$$

$$h_t = o_t \times c_t \tag{2.6}$$

In the back propagation process, loss of the network at each time step is calculated by Eq. (2.7). Total system loss from time 1 to time T is given by $L_{(t)}$ in Eq.

(2.8). Optimized weights and bias need to be find in order to minimize $L_{(t)}$. Gradient descent algorithm is typically In BP process. In gradient descent algorithm, partial derivatives of all the weights and bias, including $W_i, W_f, W_o W_g, b_i, b_f, b_o, b_g$, need to be calculated by the derivative chain rule. Eq. (2.9) to Eq. (2.12) shows how all the partial derivatives of weights are calculated, bias are calculated in a similar way. In the equation, T is the total number of time steps, M is the total number of LSTM blocks. Learning rate (α) represents the learning speed, the parameters are updated by adding the product of learning rate and parameter derivatives.

$$l_{(t)} = (h_t - y_t)^2 \tag{2.7}$$

$$L_{(t)} = \sum_{t=1}^{T} l_{(t)} \tag{2.8}$$

$$\frac{dL}{dw_o} = \sum_{t=1}^{T} \sum_{m=1}^{M} \frac{dL_{(t)}}{dh_{m(t)}} \frac{dh_{m(t)}}{do_{m(t)}} \frac{do_{m(t)}}{dw} \tag{2.9}$$

$$\frac{dL}{dw_i} = \sum_{t=1}^{T} \sum_{m=1}^{M} \frac{dL_{(t)}}{dh_{m(t)}} \frac{dh_{m(t)}}{dc_{m(t)}} \frac{dc_{m(t)}}{di_{m(t)}} \frac{di_{m(t)}}{dw} \tag{2.10}$$

$$\frac{dL}{dw_f} = \sum_{t=1}^{T} \sum_{m=1}^{M} \frac{dL_{(t)}}{dh_{m(t)}} \frac{dh_{m(t)}}{dc_{m(t)}} \frac{dc_{m(t)}}{df_{m(t)}} \frac{df_{m(t)}}{dw} \tag{2.11}$$

$$\frac{dL}{dw_g} = \sum_{t=1}^{T} \sum_{m=1}^{M} \frac{dL_{(t)}}{dh_{m(t)}} \frac{dh_{m(t)}}{dc_{m(t)}} \frac{dc_{m(t)}}{dg_{m(t)}} \frac{dg_{m(t)}}{dw} \tag{2.12}$$

$$w_o + = \frac{dL}{dw_o} \times \alpha \tag{2.13}$$

$$w_i + = \frac{dL}{dw_i} \times \alpha \tag{2.14}$$

$$w_f + = \frac{dL}{dw_f} \times \alpha \tag{2.15}$$

$$w_g + = \frac{dL}{dw_o} \times \alpha \tag{2.16}$$

Chapter 3

System Overview

The proposed prefetcher is attached to the last level cache (LLC) and snoops every LLC demand hit and miss (Fig. 3.1). This prefetcher relies on local delta history to predict the addresses of future memory accesses. This section provides an overview of the proposed prefetcher architecture.

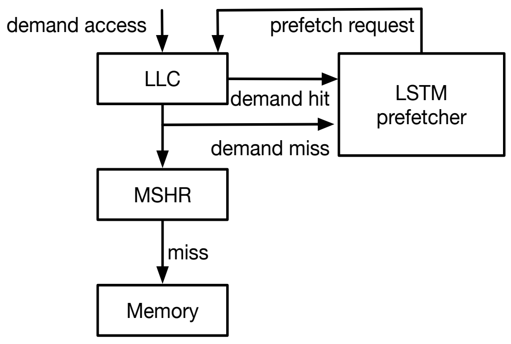

FIGURE 3.1: LSTM prefetcher in memory hierarchy

LSTM prefetcher predicts within the OS page boundary. It uses an offset table to hold the offset of the first access within a page, and a page table to record the local delta history. Offset table is indexed by page offset, and each entry stores a delta value associated with the page offset. Page table holds the recent delta histories and page numbers for different OS pages. Each page table access searches all of the page table entries to check if there is a page number match. An example page table entry is shown in Fig. 3.2. Each entry has six data fields: 1) page offset of the previously accessed address (preAddr), 2) encoded page number (pageSign) that forms part of the system input in delta prediction process, 3) four lasted delta values (pre4Delta), 4) four page offsets of the recently issued prefetches (pre4FetchAddr), 5) accuracy bit that helps to control the number of multi-degree prefetches, 6) state information for the local delta sequence is stored in the preState field. We choose to store four previous deltas because experiments show that history of the length of four can provide enough information for accurate predictions.

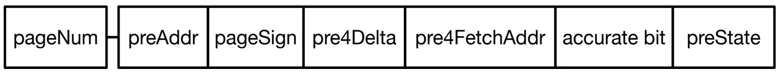

FIGURE 3.2: Page table entry

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, page table will be updated. In Fig. 3.3, hardware components (1), (2), (3), (4), (5), (6), (7) will be used for delta prediction, and (8), (7), (6), (4), (3) will be used for system training.

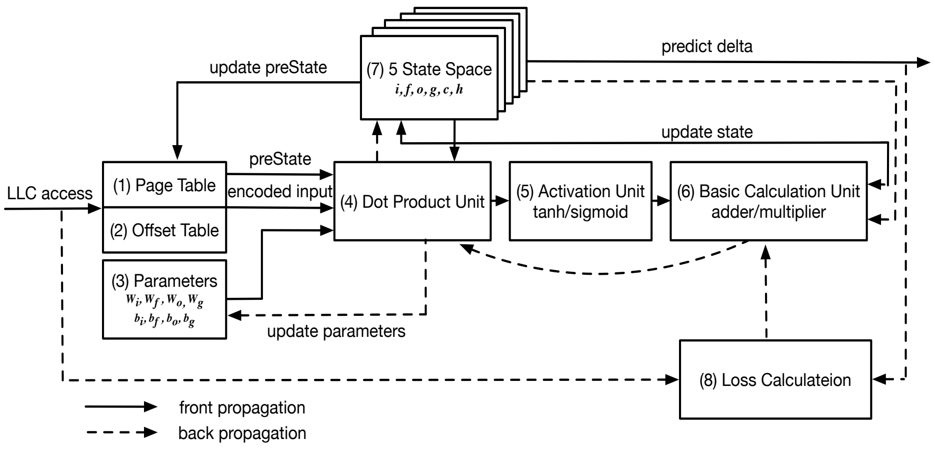

FIGURE 3.3: Internal architecture of LSTM prefetcher

In order to implement LSTM algorithm introduced in Section 2.3, a LSTM data path is built in hardware, which includes LSTM storage, run-time state storage space and functional units. Although local history for each OS page is stored separately in page 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. 3.3(3)), which are initiated to randomly generated values between -10^{-5} and 10^{-5}.

Similar to VLDP, the proposed LSTM prefetcher stores the history delta patterns in a pattern history table, and uses local delta histories as the indexes to search the matching string. LSTM algorithm provides a different mechanism for the pattern storing and searching process than VLDP. A detailed explanation for this mechanism will be discussed in 6.2. In principle, LSTM network remembers all of the history patterns by training the models and updating the parameters, and it makes prediction by using 5 delta histories (four stored in page table, one calculated on access) together with their page sign as the system inputs to predict the next address. The five state space (Fig. 3.3(7)) is the runtime storage space to store all the state value vectors which have been calculated during five time steps. Those vectors need to be stored because they will used to update parameters sequentially after the delta prediction process. However, unlike parameters which will be hold 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 parameters update.

In order to calculate the state vectors and to update the parameters, serval functional units are required. Dot Product Unit (Fig. 3.3(4)) is used for matrix multiplication in Eq. (2.1-2.4, 2.10). The Activation Unit (Fig. 3.3(5)) applies tanh and sigmod activation functions to calculate the values for state vectors i, f, o, g in Eq. (2.1-2.4). The Basic Calculation Unit (Fig. 3.3(6)) includes adders and multipliers, which help satisfy the rest of the computational requirements. The Loss Calculation Unit (Fig. 3.3(8)) is part of the basic calculation unit, it is shown separately in Fig. 3.3 because the loss function is only calculated for the training process. Chapter 5 will discuss the organizations and implementations of these functional units in details.

Chapter 4

LSTM Prefetcher Operation

Each basic prediction process is activated by a meaningful LLC access and includes two steps: delta prediction and system training. Page table is updated after the training process. In addition to predicting the next address, LSTM prefetcher can issue more prefetches based on prediction results to be more aggressive. This section will discuss two operation mode of the proposed LSTM prefetcher.

4.1 Predict The Next Address

4.1.1 Meaningful LLC Access

Although LSTM prefetcher checks every LLC access, it only activate the prediction process on meaningful LLC accessed, which captures the LLC demand misses as if no prefetch operation happens. A meaningful LLC access is either a demand miss or a demand hit on a previously prefetched cache block. When a new access enters the LSTM prefetcher, page number is used to search the page number field of the page table. If the access misses in the page table and is a meaningful access, a new entry will be added. The least recently used entry will be evicted if the page table is full. If hit in the page table and the offset matches one of the pre4FetchAddr stored in page table, this access will be considered as a meaningful access, which will activate a prefetch. Since the fetched address are stored in the order they may be used, four fetched addresses are enough.

4.1.2 Delta Prediction

LSTM prefetcher use a similar offset table as the offset table in the VLDP [12] to predict the first access to a page. The page offset of the first access is 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 hit in the offset table, the delta stored with the offset will be used to calculate the prefetch address.

On a meaningful LLC access, if there is a page table hit (which means the page has been accessed at least for the second time), a delta value will be calculated using the current offset and the offset of the previous access in this page. The 4 history deltas, current calculated deltas and the pageSign are encoded and input to the LSTM. The encoding scheme will be introduced in section 5.1.

The prediction process is also known as the front propagation process in LSTM algorithm, which includes five time steps in the proposed prefetcher. Inputs are given to the network in different time steps one by one in sequence. 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)). 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 pageSign will be used as the predicted delta value. This process is shown as the solid lines in Fig. 3.3. A detailed figure unfold in time is shown in Fig. 4.1.

When a page is accessed the first time, there is no previous output vector (c_{t-1}) or cell state vector (h_{t-1}). These two vectors are both set to zero when calculating the first set of state value vectors. Thus, preState is initialed with two zero vectors, and they will only be updated until the 5th access in this page. When the delta prediction process completes for the 5th access, the state value vectors c_1 and h_1 stored in the first state space will be recorded to the page table as preState, and everything else in the state space will be flushed. At the 6th access, first delta in this page is removed, but preState is holding the information of the first delta and will be used as c_0 and h_0 for the first time step calculation in 6th access. At each the following accesses, delta sequence stored in page table is shift left one. The oldest delta is removed and preState that holds the information of the evicted delta will be used in calculation.

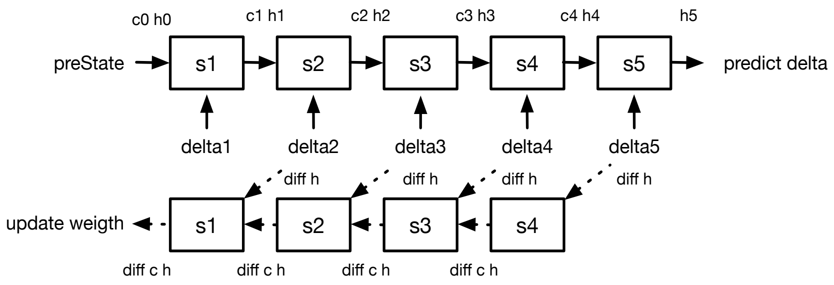

FIGURE 4.1: Time steps for delta prediction and system training

4.1.3 System Training

System Training also known as the back propagation process in LSTM algorithm (imaginary line shown in Fig. 3.3 and Fig. 4.1). This process minimize the system loss by updating weights, so that the trained network model can best predict the delta sequence.

Loss of the system comes from the difference between predicted delta and the correct delta at each time. In the example shown in Fig. 4.1, five system output vectors are generated and stored in state space after the delta prediction process. 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. In the computation process, loss at each time step is required, and derivatives of the output vectors (h) and the cell state vectors (c) from previous state space are computed. The partial derivatives of the parameter matrices are computed, and the parameters are updated to minimize the loss [19]. 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.

4.1.4 Page Table Update

When a page table entry validate for the first time, an uniq pageSign related to entry number will be given as table index, and preAddr in the page table are updated. After the second access, deltas will be computed for each access to the same page, and the latest four deltas are recored in the pre4Delta list. If a prediction is made, the predicted offset will be shifted to the pre4FetchAddr. The accuracy bit is a one-bit flag indicating whether any of the most four recent predictions are correct or not. It's updated by comparing the address of the upcoming LSTM access with the pre4FetchAddr. If any of the previous prefetch matches, the accuracy bit will set to one. If none of the previous prefetch matches, the accuracy bit will set to zero. preState can only be updated until the fifth access. When a page entry is evicted, all of the data will be discarded except pageSign.

4.2 Multi-Degree Prefetch

The proposed prefetcher can predict multiple addresses upon one LLC access by taking the predicted addresses as the new histories for high-degree prefetches. Multi-degree prefetch for LSTM prefetcher only happens on LLC hit. If a meaningful LLC hit happens and the accuracy bit for this page entry is already set to 1, this means the current prediction is on the right path, then three more prefetch requests will be issued to catch the opportunity for timely prefetches. But once a multi-degree prefetch is issued, before the accuracy bit is set to zero, no more multi-prefetch can be issued in this page.

When multi-degree prefetch happens, after the normal prediction output is computed, state vectors c and h generated at the last time step will be used as input to calculate the next delta. Fig. 4.2 shows an example of multi-degree prefetches. Since the delta prediction process is a sequential process, when degree depth increase, the prediction latency will increase too. However, the extra prediction process can be overlapped with the training process, so the total process time for each access will not increase. No extra space is needed for implementing multi-degree prefetch. This is because the calculation process is sequential, and same state space can be shared among the additional time steps.

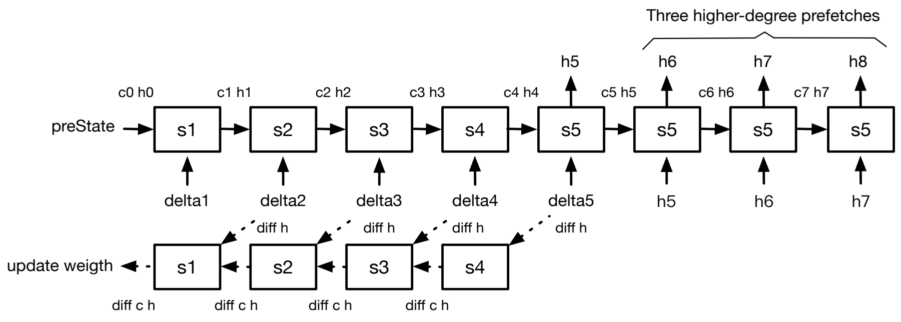

FIGURE 4.2: Multi-degree prefetch

4.3 Leveraging Next-line Prefetcher to Accelerate Prefetches

In many applications, +1 is a commonly seen delta. When predicting +1 patterns, next-line prefetcher is the most efficient approach with low cost and easy implementation. However, next-line prefetcher can not predict complex delta patterns. In the contrast, LSTM prefetcher suffers from complex calculation and long latency in predicting simple patterns. In order to efficiently predict both simple and complex patterns, a combined design is proposed.

In the combined design, a simple next-line prefetcher is implemented at the LLC level and monitors the previous four delta patterns stored in the LSTM prefetcher. If two continues +1 delta is seen, then the system assumes that there will be good spatial locality around this area, and the next-line prefetcher will be activate. It will generate three prefetch requests all with +1 delta in order to utilize the spatial locality. In the mean while, the LSTM prefetcher is still running the prediction process and getting trained. Because LSTM prefetcher needs continues trace to make future prediction. After the the long latency for LSTM prefetcher, if the predicted value is not +1, the LSTM prefetcher will issue the new prefetch, because this prediction is considered to be more accurate than the next-line prefetcher. If the predicted value is +1, no repeated prefetches need to be issued and the prediction is accelerated by the next-line prefetcher. An additional two-bit saturating counter is required by the next-line prefetcher to calculate the global accuracy for itself, if the global accuracy is below a threshold, then the next-line prefetcher will be turned off to avoid further pollution.

A simple example is given to show when accelerate prefetch works. Consider pattern (a, 1, 1, 1, 1, 1, b) continues repeating in a single page, while (a, 1, 1, 1, 1, 1) is trained to predict (b), prediction of the last three 1 in the +1 string will get accelerate. In this way, more prefetch requests can be issued more timely, and catch more hit opportunity.

Table based prefetcher can predict +1 pattern in a short time (longer than next-line but short than LSTM prefetcher), and it works well when continues +1 delta and complex delta pattern shows in the different pages. Because in this case, the +1 pattern is always holding one DPT entry during the entire application running time. However, when patterns shown in the above example exists, since cascade table has limited length, the +1 entry in the DPT table will be messed up by other patterns. Because (1,1,1 -> 1), (1,1,1 -> a), (1,1,1 -> b) exists simultaneously, and keep kicking out each other in the same entry. So combined LSTM and next-line prefetcher design can be more efficient than traditional table based design in predicting this kind of patterns.

Chapter 5

Implementation of the LSTM Prefetcher

For neuron network algorithm, hardware implementation is a traditional challenge, since this kind of approach generally requires large amount of calculation and huge storage space. In this section, data encoding mechanism and different hardware design choice are analyzed to show that the given LSTM algorithm can be implemented with affordable hardware overhead, and acceptable latency.

5.1 Delta Representation

System input for LSTM network consists of two parts, one part is pageSign, the other part is delta. The reason of having two parts is to provide opportunities for weight sharing and to help reducing the aliasing problem. The two parts are encoded in binary representations separately. Since delta could be a negative value, two's complement representation is used. System output is the predicted binary results for pageSign and delta, output without pageSign part is used to predict the next address.

5.2 Intermediate

In addition to input and output, intermediate data required by LSTM include parameters and state vectors. Using floating point to store the intermediate data will lead to a high accuracy. However, floating point operation requires complex functional units.

The alternative is to use the fix-point representation. An offline LSTM network experiment shows that the dynamic range of this application is not very large: the largest value for the intermediate data is 32, and the smallest value is 0.001. A 16 bit fix point representation can be used with a 6 bit signed integer part and a 10 bit fractional part.

5.3 Functional Units

5.3.1 Total Computation Needs

When certain input is given, LSTM network requires computation to get the output results. The computation process on each meaningful LLC access comprised two parts, delta prediction and system training. Functional units can be shared between these two processes at different time steps.

Delta prediction is a sequential process with five time steps (at each time steps a different input is given). After the delta prediction process, a prediction result is generated by the network and the prefetch requests can be sent. On the critical path for each time step, one dot product computation, two additions, two multiplications, and 1 activation are required. Every elements in the same matrix or vector are processed in parallel. So the total latency for the prediction process is

$$5 \times (T_{dot} + 2 \times T_{add} + 2 \times T_{multiply} + T_{activation})$$

For the training process, losses are calculated during four sequential time steps to update the network parameters. At each time step, one dot product computation, four additions, and four multiplications are needed. The total latency for the training process is

$$4 \times (T_{dot} + 3 \times T_{add} + 4 \times T_{multiply})$$

Trade-offs exist between latency and hardware overheads. Parallel processing reduces the latency while increase hardware overheads. Calculations within a delta prediction or training process are sequential among different time steps. However, individual operation (e.g, vector operations) can have data-level parallelisms. In addition, upon observing an LSTM access, the prediction and training can be processed in parallel. An example is given in the following paragraph to demonstrate both the sequential and the parallel approaches. In this example, k LSTM blocks are assumed in the network. Dot product computation, addition, and activation all exhibit a one-cycle latency, while multiplications has two-cycle latency.

For the sequential design, the new delta prediction process starts after the previous training process is done, and the state is updated. Upon receiving a new LSTM access, if the previous training have completed, the total latency will be 40-cycle prediction latency. If the previous training have not completed, the prediction can wait upon to 48 cycles to start. The sequential implementation does not allow more than one prediction in the system. If two access arrive within 40 cycles, the second one is dropped. For the parallel design, delta prediction process can be overlapped with the previous training process, so the latency will always be 40 cycles. However, In the parallel implementation, all of the states need to have a second copy, so there will be more storage overheads. Hardware can be shared in both approaches, only one dot product unit and one activation unit are needed, for the basic computational units, $6 \times k$ multipliers and $4 \times k$ adders are needed to maximize data level parallelisms at the operation level.

5.3.2 Implementing Activation Function

The activation function transforms an input to an output using a given function. LSTM network uses two activation functions, sigmoid function and hyperbolic tangent function tanh (Fig. 5.1). Multiple mechanisms have been proposed for implementing these two functions in hardware [26] [27] [28] [29].

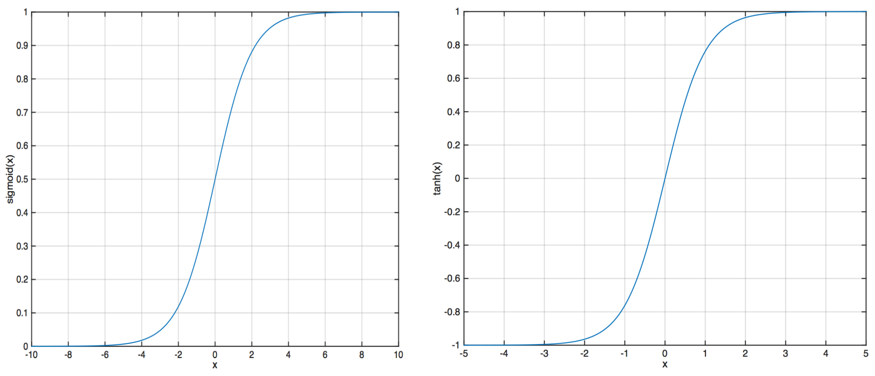

FIGURE 5.1: Activation function (left: sigmoid, right: tanh)

Look up table (LUT) [30] [31] [32] [26] is the most commonly used approach to implement the sigmoid and tanh functions. In this approach, 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, thus LUT is the fastest implementation as compared to other techniques. However, in order to get better accuracy, large storage space are required for LUT implementation. Conventional LUT design requires more than 1KB memory space. A recent optimized LUT design[33] decreases this size to less than 15B, while having the maximum error distance less than 0.02.

Piecewise linear approximation (PWL) and piecewise nonlinear approximation are two alternative implementations [34] [35]. These two approaches use some simple functions to approximate the complex activation functions. The advantage is they do not have large storage overhead. However, the computations need longer latency than LUT based implementation.

For LSTM prefetcher design, since latency is the most critical parameter directly related to prefetcher performance, we choose to use LUT to implement both sigmoid and tanh funcion.

5.3.3 Implementing Matrix-Vector Multiplication

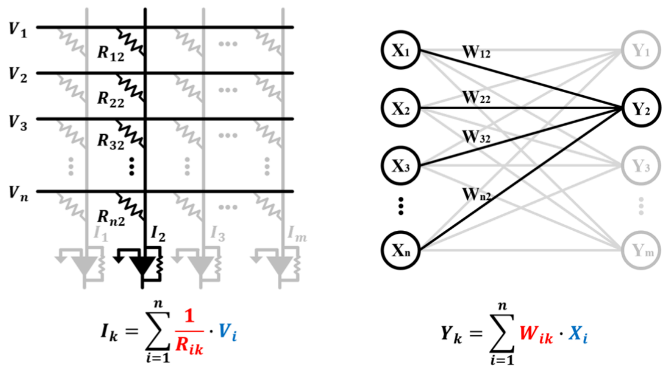

FIGURE 5.2: An example of mapping vector-matrix multiplication to RRAM crossbar array (adapted from [1])

When doing matrix-vector multiplication, the emerging nonvolatile memory (NVM) crossbar array shows a great potential. As shown in Fig. 5.2, the function of calculating the current in a crossbar array has similar format as a matrix-vector multiplication. If the input vector are properly mapped to the input voltages and the weight matrix is programmed to the resistors, the matrix-vector multiplication can be done by simply sampling the current flowing in each array. Thus lead to a fast efficient implementation of matrix-vector multiplication.

The resistance is programmable, before getting the result of matrix-vector multiplication, resistance array need to be set proportional to the correspond weight matrix values. If this mapping process happens frequently, the latency will be long due to the long write latency of NVMs. Luckily, for LSTM network, since pattern exists during the entire application trace, after the states are converged, the weight matrix will not change. So the average latency will be acceptable. By using NVM-based crossbar, dot product computation can be done within a single clock cycle.

Chapter 6

Case Study

In this section, our design configurations for LSTM network is given, and hardware overheads for this proposed design is calculated in section 6.1. An offline case study is conducted in order to further analyze the prefetching mechanism. Results of the offline case study are shown in section 6.2.

6.1 Prefetcher Configurations

We simulated LSTM prefetcher with one 64-entry offset table, one 64-entry page table, and one LSTM network. We choose 4KB os page size and 64B cache block size. 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 pageSign. The total input length is 13 bits.

In the LSTM network, 13 LSTM blocks form a hidden layer, which connects to 13 input and 13 output nodes. The network structure is shown in Fig. 6.1. There are 4 weight matrices (W_g, W_i, W_f, W_o) related to each gate within each block, each with a size of 13*26 bits. Each gate also requires 4 bias vectors (b_g, b_i, b_f, b_o), each of which has 13 bits. We choose history length of 5, thus, 5 history deltas are used as inputs in 5 time steps to make the prediction. Each of the five state spaces contains 6 state vectors (i, f, o, c, s, h) with a size 13. Data stored in the matrices are all 32 bit floating point values. The learning rate is set to 0.2, which gives the best performance for the system. Total storage overhead for this design is calculated in Table 6.1. With a 16-bit fix-point inner data representation, total storage overhead for this design could be decreased to 4.4KB. In this design, we choose the sequential approach for delta prediction and system training.

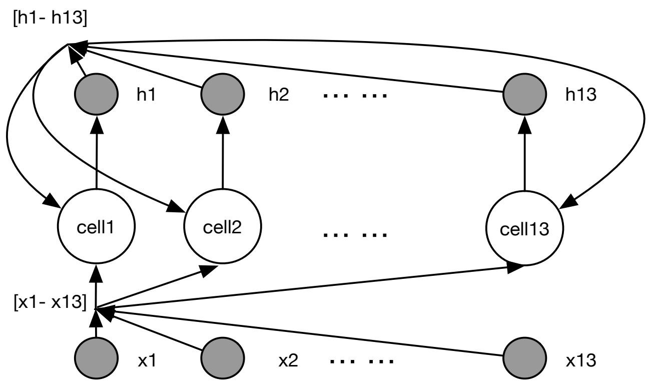

FIGURE 6.1: Network structure

TABLE 6.1: LSTM Prefetcher Storage Overheads

Component	**size in bit**	**size in Byte**
offset table	$64 \times (6 + 7)$	104B
page table	$64 \times (20 + 6 + 4 \times 7 + 4 \times 6 + 1 + 2 \times 13)$	840B
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		7.9KB

6.2 Offline Result and analysis

Although the idea of using delta history to predict the address of future memory access for LSTM is similar with VLDP [12] and signature path prefetcher (SPP) [21], LSTM algorithm is different from traditional table based algorithms. We conducted an offline study to analyze the performance of LSTM prefetcher on three selected traces.

We build both LSTM prefetcher and VLDP prefetcher offline without multi-degree prefetching or leveraging the next-line prefetches. Generated traces are used

TABLE 6.2: Comparison Between VLDP and LSTMP Prediction Accuracy

Prefecher / Trace	$a_1a_2a_3 \ldots a_{65}$	**a[^a]{20,20}b[^a]{20,20}c[^a]{20,20}**	**a{6,6}[^a]**
VLDP Accuracy	0%	75%	26.6%
Num of issued prefetch	0	17991	46979
LSTMP Accuracy	98.6%	83.5%	61%
Num of issued prefetch	0	17989	46991

to test accuracies of these two prefetchers. The traces are represented by regular expressions. To ensure fairness, the 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 LSTM prefetcher. 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 the table 6.2.

The first trace consists of a long repeating pattern ($\boldsymbol{a_1}$, $\boldsymbol{a_2}$, $\boldsymbol{a_3}$...$\boldsymbol{a_{65}}$), VLDP can not make any prediction because the useful sub-string information will be flushed before it is encountered again. For example, when ($\boldsymbol{a_1}$, $\boldsymbol{a_2}$) get accessed, sub-string ($\boldsymbol{a_1}$, $\boldsymbol{a_2}$) will be recorded to DPT table. However, in the following accesses, ($\boldsymbol{a_2}$, $\boldsymbol{a_3}$), ($\boldsymbol{a_3}$, $\boldsymbol{a_4}$) ... ($\boldsymbol{a_{64}}$, $\boldsymbol{a_{65}}$) will be recorded in the DPT table. Since the DPT table has limited number of entries, ($\boldsymbol{a_1}$, $\boldsymbol{a_2}$) will be flushed before encountering the next ($\boldsymbol{a_1}$, $\boldsymbol{a_2}$) pattern. As a result, VLDP will lose all of the prediction opportunities. The result shows VLDP issues zero prefetch, and the accuracy is zero. However, for the LSTM prefetcher, the network remembers the entire sequence directly. After training properly, any delta in this sequence given the network inputs will result in an output which is the exact next delta in the same sequence. These feature gives LSTM prefetcher the potential in remembering long sequence in a simple way, thus leads to a higher accuracy. The test results show that LSTM can achieve 98.9% accuracy in predicting the first trace.

The second trace consists of pattern a[ˆa]{20,20}b[ˆa]{20,20}c[ˆa]{20,20}, which means delta (a) is followed by another delta other than (a), these pattern repeats 20 times, then delta (b) is followed by another delta other than a, which is also repeats 20 times. An example could be (a, b, a, c, a, d ... b, b, b, c, b, d ... c, b, c, c, c, d ... a, b, a, c, a, d ...). In this kind of patterns, (a, b) only lead to (a) in the entire sequence. However, since next (a, b -> a) pattern only appears after a long time, previous information is already flushed since VLDP only has limited entry. Thus prevent it from making correct prediction. Our experiment shows a 27.4 % accuracy for VLDP on this trace. For LSTM prefetcher, 63.7 % accuracy can be achieved.

The third trace includes a repeating pattern a{6,6}[ˆa], which means six deltas (a) is followed by a delta different from a, for example (a, a, a, a, a, a, b, a, a, a, a, a, a, c, a, a, a, a, a, a, d ...) is one of such traces, VLDP shows a 77.7 % accuracy, whereas LSTM prefetcher shows a 85 % accuracy. LSTM works better because VLDP suffers from the aliasing problem. Limited history length of the DPT table prevent VLDP from get enough history information to make correct predictions. Since delta history (a, a, a, a, a) can lead to different next deltas (a), (b), (c), (d) ... , no matter how many times these patterns appears, VLDP will always lose the chance to correctly predict the delta other than (a). What's more, entry (a, a, a, a, a -> a) can be messed up too, which leads to more performance degradation. LSTM prefetcher has aliasing problem too, which leads to its mis-prediction on (b), (c), (d)..., however, it can correctly predict all of the (a)s. This is because LSTM prefetcher is better at tolerating noises.

Using previous state of an newly evicted delta as one of the prediction input is another specific design choice made for the LSTM prefetcher. Since local delta history shift left once on each access, and a certain delta will be evicted after four following delta 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 used in the LSTM prefetcher is more than five. This property can help increase accuracy.

When making predictions, table based schemes use local delta string to search the global table, while LSTM prefetcher needs to do computation to calculate the state vectors. Calculation takes longer time than searching the tables. However, LSTM prediction latency can fit in most of the memory access intervals.

While having the advantages mentioned above, LSTM mechanism has it is own disadvantages too. Warm up time is one major draw back. 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. This problem can be alleviated by the global pattern 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 several 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 lose chances as well. This side effect of LSTM prefetcher will get amplified when many pages are access in an interleaved way and a certain page is frequently evicted and added.

Another disadvantage shown when the aliasing problems frequently appear in a single page. In this case, LSTM prefetcher will always use the latest observed sequence to make the prediction. The worst case happens when patterns like (a, a, b) and (a, a, c) interleaves without any pattern. When this situation happens, LSTM prefetch will always be wrong in predicting different deltas (b and c) for both sequences. One possible way to help this scenario is to use the offset other than the delta as inputs, to provide more information to avoid aliasing, which remains as one of our future works.

Bibliography

[1] Wm A Wulf and Sally A McKee. “Hitting the memory wall: implications of the obvious”. In: *ACM SIGARCH computer architecture news* 23.1 (1995), pp. 20–24.

[2] Hyesoon Kim Jaekyu Lee and Richard Vuduc. “When prefetching works, when it doesnot, and why”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 9.1 (2012), p. 2.

[3] Alan Jay Smith. “Sequential program prefetching in memory hierarchies”. In: *Computer* 11.12 (1978), pp. 7–21.

[4] Norman P Jouppi. “Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers”. In: *Computer Architecture, 1990. Proceedings., 17th Annual International Symposium on*. IEEE. 1990, pp. 364–373.

[5] Subbarao Palacharla and Richard E Kessler. “Evaluating stream buffers as a secondary cache replacement”. In: *Computer Architecture, 1994., Proceedings the 21st Annual International Symposium on*. IEEE. 1994, pp. 24–33.

[6] Santhosh Srinath et al. “Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers”. In: *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*. IEEE. 2007, pp. 63–74.

[7] Seth H Pugsley et al. “Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers”. In: *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*. IEEE. 2014, pp. 626–637.

[8] Pierre Michaud. “A best-offset prefetcher”. In: *2nd Data Prefetching Championship*. 2015.

[9] Yasuo Ishii, Mary Inaba, and Kei Hiraki. “Access map pattern matching for high performance data cache prefetch”. In: vol. 13. 2011, pp. 1–24.

[10] Stephen Somogyi et al. “Spatial memory streaming”. In: *ACM SIGARCH Computer Architecture News* 34.2 (2006), pp. 252–263.

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

[12] Manjunath Shevgoor et al. “Efficiently prefetching complex address patterns”. In: ACM. 2015, pp. 141–152.

[13] Zichao Yang et al. “Stacked attention networks for image question answering”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 21–29.

[14] Dario Amodei et al. “Deep speech 2: End-to-end speech recognition in english and mandarin”. In: *arXiv preprint arXiv:1512.02595* (2015).

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

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

[17] Sam Romano and Hala ElAarag. “A neural network proxy cache replacement strategy and its implementation in the Squid proxy server”. In: *Neural computing and Applications* 20.1 (2011), pp. 59–78.

[18] Demetri Psaltis, Athanasios Sideris, and Alan A Yamamura. “A multilayered neural network controller”. In: *IEEE control systems magazine* 8.2 (1988), pp. 17–21.

[19] Sepp Hochreiter and Jürgen Schmidhuber. “Long short-term memory”. In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[20] Xiangyao Yu et al. “IMP: Indirect memory prefetcher”. In: *Proceedings of the 48th International Symposium on Microarchitecture*. ACM. 2015, pp. 178–190.

[21] Jinchun Kim et al. "Path confidence based lookahead prefetching". In: *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE. 2016, pp. 1–12.

[22] Saami Rahman et al. "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. 2015, pp. 383–389.

[23] Shih-wei Liao et al. "Machine learning-based prefetch optimization for data center applications". In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM. 2009, p. 56.

[24] John Cavazos and Darko Stefanovic. "Adaptive Prefetching using Neural Networks". In: Citeseer, 1997.

[25] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. "Learning long-term dependencies with gradient descent is difficult". In: *IEEE transactions on neural networks* 5.2 (1994), pp. 157–166.

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

[27] Che-Wei Lin and Jeen-Shing Wang. "A digital circuit design of hyperbolic tangent sigmoid function for neural networks". In: *Circuits and Systems, 2008. ISCAS 2008. IEEE International Symposium on*. IEEE. 2008, pp. 856–859.

[28] Dr ShrugalVarde and RichaUpadhyay NishaSarwade. "Hardware Implementation Of Hyperbolic Tan Using Cordic On FPGA". In: ().

[29] Karl Leboeuf et al. "High speed VLSI implementation of the hyperbolic tangent sigmoid function". In: *Convergence and Hybrid Information Technology, 2008. ICCIT'08. Third International Conference on*. Vol. 1. IEEE. 2008, pp. 1070–1073.

[30] A Muthuramalingam, S Himavathi, and E Srinivasan. “Neural network implementation using FPGA: issues and application”. In: *International journal of information technology* 4.2 (2008), pp. 86–92.

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

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

[33] Pramod Kumar Meher. “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. 2010, pp. 91–95.

[34] MT Tommiska. “Efficient digital implementation of the sigmoid function for reprogrammable logic”. In: *IEE Proceedings-Computers and Digital Techniques* 150.6 (2003), pp. 403–411.

[35] Ming Zhang, Stamatis Vassiliadis, and Jose G. Delgado-Frias. “Sigmoid generators for neural computing using piecewise approximations”. In: *IEEE transactions on Computers* 45.9 (1996), pp. 1045–1049.

Vita

Yuan Zeng received her bachelor degree in electronic engineering at Beijing Jiaotong University, Beijing, China in 2015. She started pursuing the master degree in electrical engineering at Lehigh University, Bethlehem, PA in 2015. Her research areas is computer architecture.