DeepPrefetcher: A Deep Learning Framework for Data Prefetching in Flash Storage Devices

Gaddisa Olani Ganfure, *Student Member, IEEE*, Chun-Feng Wu, *Graduate Student Member, IEEE*, Yuan-Hao Chang, *Senior Member, IEEE*, and Wei-Kuan Shih, *Member, IEEE*

***Abstract*—In today's information-driven world, data access latency accounts for the expensive part of processing user requests. One potential solution to access latency is prefetching, a technique to speculate and move future requests closer to the processing unit. However, the block access requests received by the storage device show poor spatial locality because most file-related locality is absorbed in the higher layers of the memory hierarchy, including the CPU cache and main memory. Besides, the utilization of multithreading results in an interleaved access request making prefetching at the storage level more picky using existing prefetching techniques. Toward this, we propose and assess DeepPrefetcher, a novel deep neural network inspired context-aware prefetching method that adapts to arbitrary memory access patterns. DeepPrefetcher learns the block access pattern contexts using distributed representation and leverage long short-term memory learning model for context-aware data prefetching. Instead of using the logical block address (LBA) value directly, we model the difference between successive access requests, which contains more patterns than LBA value for modeling. By targeting access pattern sequence in this manner, the DeepPrefetcher can learn the vital context from a long input LBA sequence and learn to predict both the previously seen and unseen access patterns. The experimental result reveals that DeepPrefetcher can increase an average prefetch accuracy, coverage, and speedup by 21.5%, 19.5%, and 17.2%, respectively, contrasted with the baseline prefetching strategies. Overall, the proposed prefetching approach surpasses other schemes in all benchmarks, and the outcomes are promising.**

***Index Terms*—Data prefetching, deep learning, flash storage devices, logical block address (LBA).**

Manuscript received April 18, 2020; revised June 12, 2020; accepted July 6, 2020. Date of publication October 2, 2020; date of current version October 27, 2020. This work was supported in part by Academia Sinica under Grant AS-CDA-107-M05, and in part by the Ministry of Science and Technology under Grant 109-2218-E-002-019, Grant 107-2923-E-001-001-MY3, Grant 108-2221-E-001-001-MY3, and Grant 108-2221-E-001-004-MY3. This article was presented in the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems 2020 and appears as part of the ESWEEK-TCAD special issue. *(Corresponding authors: Gaddisa Olani Ganfure; Yuan-Hao Chang.)*

Gaddisa Olani Ganfure is with the Social Networks and Human-Centered Computing, Taiwan International Graduate Program, Institute of Information Science, Academia Sinica, Taipei 11529, Taiwan, and also with the Institute of Information Systems and Applications, National Tsing Hua University, Hsinchu 300, Taiwan (e-mail: gaddisaolex@gmail.com).

Chun-Feng Wu and Yuan-Hao Chang are with the Institute of Information Science, Academia Sinica, Taipei 11529, Taiwan (e-mail: cfwu@iis.sinica.edu.tw; johnson@iis.sinica.edu.tw).

Wei-Kuan Shih is with the Department of Computer Science, National Tsing Hua University, Hsinchu 300, Taiwan (e-mail: wshih@cs.nthu.edu.tw).

Digital Object Identifier 10.1109/TCAD.2020.3012173

I. Introduction

IN THE big-data era, the growing flood of data demands for larger storage space. Due to the cheaper unit cost and nearly no leakage power, industries are deploying solid-state drives (SSDs) as storage devices in current systems. With avoiding SSDs becoming the performance bottleneck, the SSD buffer (e.g., DRAM or SRAM) inside the SSD is usually used to cache the data, so as to hide the slow access latency incurred by accessing the NAND flash chip. Data prefetching is one of the potential solutions to predict and move the data from the NAND flash chip to the SSD buffer to improve the device responsiveness. However, it is challenging for designing effective data prefetcher for storage devices. The reason is that most file-related locality is absorbed in the higher layers of the memory hierarchy, and thus the access patterns received by the storage devices show weak-locality and are hard to be predicted. Behavior-based prefetcher seems to work well for some applications; however, sequential access patterns are not the dominant disk access pattern for some applications, like volume visualization or digital video playbacks [1], [2]. In such domains, strategies that usually exploit the spatial locality is not useful in predicting storage access patterns.

Aiming to predict weak-locality access patterns, several previous works proposed learning-based prefetchers to learn from the pattern of storage requests with the support of machine learning. Based on our observations, the main challenges of designing learning-based prefetchers for the storage devices are in twofold. First, the semantic gap between the systems and the storage devices hinders various access behaviors from being sent inside the devices, and thus the most commonly used training data are the logical block address (LBA) sequences. Second, the model complexity is positively correlated with the varying degree of training data. The overall numbers of storage unique LBAs (or the variance degree of LBAs) are hundreds to thousands of times higher than that of the main memory or CPU cache. Therefore, the complexity of feature extractions for learning storage access patterns is more challenging than that of the access patterns retrieved from the main memory or CPU cache. These challenges would result in the learning complexity issue, and this issue becomes more serious when the size of storage devices grows. Motivated by this observation, this work proposes a deep learning way to tackle the learning complexity problem by providing new features that is vital to reduce the variance degree of training data so that the learning-based prefetchers potential can be unleashed even in the storage devices.

Data prefetcher is a widely used approach to close the performance gap between two devices with different access latency. Data prefetcher usually performs speculation for fetching the data from a slower device to a faster device with being aware of recent data patterns. Behavior-based data prefetchers are widely used in memory or file system layers due to the advantage of easy design and effective performance. The design concept of behavior-based data prefetchers is to be aware of the recent spatial or temporal access behaviors for predicting the following access behaviors. Sequential prefetcher is one of the most famous behavior-based data prefetchers, and it takes advantage of the spatial locality extracted from both process behaviors [3] and file access behaviors [4]. To be specific, the access behaviors of a normal process usually show strong spatial locality [3], and the file system usually allocates a continuous physical address for each file, which shows a strong spatial locality. Moreover, the well-known Linux readahead [5] system call adopts a sequential prefetcher to minimize the recurrence of accessing storage devices.

Even though behavior-based prefetchers are effective at predicting future accesses at the system-level, behavior-based prefetchers are not compelling for prefetching at the storage level due to the weak-locality of the LBA access patterns. To extract weak-locality features in the storage devices, a number of machine learning-based approaches are proposed, such as autoregressive integrated moving average (ARIMA) model [6], C-miner [7], and Diskseen [8]. Prefetching techniques like the ARIMA model require a long sequence with no structural change to make a correct prediction, whereas the frequent pattern mining approaches require repetitive access patterns to build the association rule, which can be reused for deciding the to-be-prefetched data. However, based on our observations, there are two main challenges in designing a learning-based prefetcher for the storage devices. First, there exists a semantic gap between storage systems (e.g., file system) and storage devices, making the learning-based prefetchers more challenging to learn useful patterns in the storage devices. Because to ensure the compatibility between storage systems and different storage devices, several abstraction layers, such as Linux block I/O layer (BIO) and flash translation layer (FTL), are applied to simplify the communication by only using `(LBAs)`. Second, the utilization of multithreading results in an interleaved access request making prefetching at the storage level more picky using existing prefetching techniques. Training machine learning models like frequent pattern mining with higher variance degrees of LBA makes the model learn very few rules, limiting the coverage of the prefetcher to conquer the miss latency. Besides, these challenges would result in the learning complexity issue, and this issue becomes more serious when the size of storage devices grows.

This work is motivated by the learning complexity concerns in adopting a learning-based prefetcher in storage devices. The "DeepPrefetcher," *a deep learning approach to predict and prefetch data in flash storage devices* is proposed to fill up the implementation gap with tacking the learning complexity. The design concept is to generate a new learning feature called "LD features or block difference." The block difference (or *LD*) is the difference between two consecutive read requests [see (1) for how the value of LD is calculated]. This new learning feature provides extra semantic information as well as lessen the degree of variance in the training data to reduce the learning complexity. Using the *LD* features as the input, the model transforms the *LD* features into a new embedded representation that is rich in context. The embedded representation undergoes the long short-term memory architecture (LSTM) to capture the hidden features in the training data. Finally, the model produces the most likely *LBA* as the output. This framework allows the proposed model to achieve better accuracy (i.e., up to 27 improvements) and coverage (i.e., up to 25 improvements) than the baseline prefetching strategies.

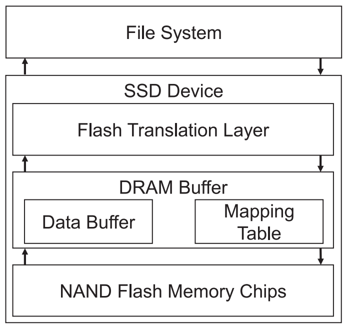

Fig. 1. Typical architecture of SSD.

The remainder of this article is structured as follows. Section II introduces SSD and related works. Observation and motivation are presented in Section III. In Section IV, DeepPrefetcher is proposed to improve prefetching accuracy and coverage. Section V discusses the evaluation result and, finally, Section VI concludes this work.

II. Background and Related Work

A. Background: SSD and Data Prefetcher

With lower unit costs and shorter access latency, flash-based SSDs are taking market share of hard disk drives (HDDs) in the storage device market. Although SSD latency is faster than that of HDD, the time for accessing SSDs still dominates the system execution time. The response time of SSD is composed of address translation latency and data movement latency [9]. As shown in Fig. 1, emerging SSDs are equipped with extra DRAM buffer for relieving the address translation overhead and the data movement latency by minimizing the frequency of accessing the NAND flash memory chip. The newly updated data can be cached in the DRAM buffer to efficiently and easily minimize the response time for dealing with write requests. *However, it is challenging to minimize the response time incurred by reading the data from the* NAND *flash memory chip.*

Data prefetching is a prominent strategy to hide the data access latency by moving the extra amounts of data that will be requested next from the flash memory chip to the SSD DRAM buffer. With current SSD designs, the continuously growing channel parallelism [10] degree reduces the overhead for reading the extra amount of data and provides the opportunity for applying a data prefetcher inside SSD. However, a data prefetcher is a double-edged sword; that is, prefetching wrong data will pollute the DRAM buffer, whereas correct prefetching will improve the latency of SSD. Thus, balancing the tradeoff between the miss coverage and accuracy is very important for a prefetch design.

B. Related Work

The prefetching concept has gained considerable attention in the research community for decades as a potential method to reduce the processor stall time [11]–[14]. The most commonly used method is called stride prefetcher, which works by triggering a prefetching task once a load access pattern and a constant stride (or pattern) were detected for several consecutive cache-misses. Stride prefetchers are implemented using a reference prediction table (RPT) [12] that stores the instruction address, program counter (PC), the last access address, and the stride [13]. Thus, to make an inference or speculation, a prefetcher can access the history from RPT to obtain the required correlations. The prefetching mechanism is activated once a pattern shows a constant stride [13]. For instance, if the last stored address is X and the detected constant stride is 2, the next address to be prefetched will be $X+2$. Although stride prefetcher is successful at capturing a constant stride that leads to cache misses, it has a few impediments. For instance, the stride is calculated and stored in RPT for two consecutive accesses [14], and thus it is difficult to detect variable strides (irregular access patterns) as in $\{1, 2, 4, 8, 16\}$.

Like prefetching at the CPU level, disk-level (or storage level) prefetching is also introduced in different works [6], [7], [15]. However, unlike CPU-level prefetching, the PC-information and other application contexts are not accessible at the storage end, which makes it challenging to adapt the CPU-level prefetcher algorithms for prefetching data at the storage level. Thus, most of the technique solely relies on modeling the `LBA` access patterns for prefetching the next request. Several well-known works are classified into two categories: 1) behavior-based and 2) learning-based prefetchers. Specifically, behavior-based prefetchers aim to decide the to-be-prefetched data by considering the data access behavior (e.g., spatial locality on LBA), and the learning-based prefetchers adopt the machine learning algorithm to predict and prefetch the data.

Sequential prefetchers are one of the representative behavior-based prefetchers [3], [4], [16] and are widely used in storage systems and database systems. Sequential prefetchers prefetch the data block whose `LBA` is $b+1$ whenever the block with `LBA` b is requested so that the time spent to read block $b+1$ can be minimized. The design concept of the sequential prefetcher is to take advantage of the file allocation where each file will be allocated to a consecutive `LBA` in the current file system. According to this design concept, sequential prefetchers are effective when devices serve I/O with strong spatial locality but show poor performance in dealing with random or interleaved requests. However, under current file system designs, most recently used file data will be cached in the page cache in the main memory, and thus, page cache absorbs most of the file access behaviors, including spatial locality. That is, most storage requests sent to storage device shows very weak spatial locality, and thus it is hard for a sequential prefetcher to show its talent.

On the other hand, learning-based prefetchers, such as frequent pattern mining [7] and ARIMA [6], are another popular class of data prefetchers. Frequent pattern mining relies on the history of the observed `LBA` patterns to predict the subsequent blocks. For instance, in C-miner [7], if the access to `LBA` $\{A, B\}$ is followed by `LBA C`, the prefetcher will prefetch block `C` whenever the pattern $\{A, B\}$ is observed assuming that enough supports are accumulated for $\{A, B\} \rightarrow \{C\}$ pairs. Different from sequential prefetchers, this approach takes both the temporal and spatial block access correlations into consideration. However, this approach has the following main issues. First, to build the association rule for frequent items, it demands large memory space to retain all access patterns of past accessed `LBAs`. Second, searching for the precise match leads to expensive computation overhead and makes it costly to deploy for real-time applications. Besides, the pattern observed for a specific sequence will not generalize to the other `LBA` access pattern and, thus, it limits the prediction capacity of this approach. The ARIMA model [6] treats data prefetching as a time series forecasting problem. The intuition of this approach is to learn the coefficients of regression from the last observation to predict the next `LBAs`. However, the correlation between access patterns is not always steady, especially when the following requests tend to be random or interleaved. Thus, it leads to higher prefetching errors if access pattern changes frequently.

III. Observation and Motivation

As discussed in Section I, data prefetching at storage level has several main challenges. For example, most requests received by the storage device shows poor spatial locality, because most file-related locality is absorbed in the higher layers of a memory hierarchy, including both CPU cache and main memory. Although some learning-based approaches are designed to capture the nonsequential (or random) access pattern, these approaches are hard to predict the unseen access patterns. The reason is that the patterns of accessed LBAs are infrequent (i.e., usually show less repetition), and the possible values of LBAs are vast, which contributes to learning complexity. Note that the learning complexity is the complexity of performing training to achieve the guaranteed level of accuracy, and it is affected by the distribution of data in a training dataset.

To validate these challenges, we perform a series of analysis on the MSR Cambridge user home directory trace [17]. Fig. 2 shows a snapshot of the trace, where the x-axis indicates the access orders parsed by the read requests and the y-axis indicates the logical block number (`LBA`) accessed by each request. The reason to only focus on the read request is that a read request will directly hurt the service time of the device whenever the requested data is not in the data buffer. However, a write request usually writes the data to the device buffer and thus may not seriously hurt the service time (or response time). The trend of accessed `LBAs` shown in Fig. 2 shows

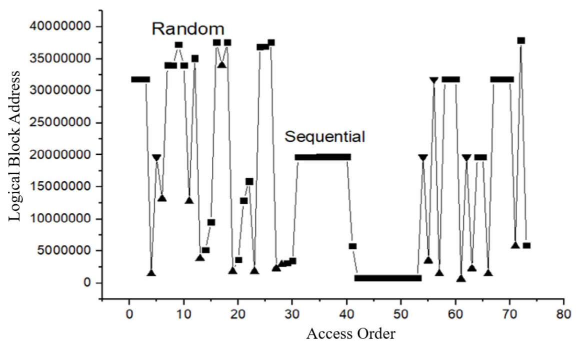

Fig. 2. Irregularity of I/O access patterns.

TABLE I
EXAMPLE OF LD FEATURE

Timestamp	LBA	LD	Timestamp	LBA	LD
t	854943	40	t_x	1476839	40
t+1	854983	16	t_{x+1}	1476879	16
t+2	854999	48	t_{x+2}	1476895	48
t+3	855047	88	t_{x+3}	1476943	88
predict(t+4)	855135	24	predict(t_{x+4})	1477031	24

Algorithm 1: LBA Preprocessor

```
Input:
T:LBA Sequence,
L:Window size,
O: Overlap percentage
Output: $X, y$
/* Input and Ouput pairs */
1  $i \leftarrow 0$ /* Index of LBA in T */
2  $j \leftarrow 0$ /* Index of subsequence */
3  $X \leftarrow \emptyset$ /* Array of features */
4  $y \leftarrow \emptyset$ /* Array of Class labels */
5  $k \leftarrow L * (O/100)$ /* k is the step */
6  while $i + L < Length(T)$ do
7      $X[j] \leftarrow T.getSubSequence(i, (i + L - 1))$
8      $y[j] \leftarrow T.getSubSequence(i + L)$
9      $i \leftarrow i + k$
10     $j \leftarrow j + 1$
11 end
12 return $X, y$
```

that the LBA access patterns are unpredictable as it randomly shifts from sequential accesses to nonsequential accesses (or random accesses) and vice versa. For example, as shown in Fig. 2, the access orders from 32 to 40 exhibit the sequential accesses, and the access orders from 60 to 66 show the random accesses. Our analysis on some MSR Cambridge Block I/O trace also shows that more than 64% of the LBA read requests are accessed only once at the time of collecting the trace. With the vast number of possible LBAs and nearly random sequence, it is hard to generate a useful model for the learning-based prefetchers.

In contrast to previous works on extracting useful features from access patterns between nearby accessed LBAs, we observed that if a sequence of LBA requests is divided into several sections, two different sections are possible to be accessed by a similar access pattern. New features can be extracted from the observed access pattern by monitoring the change (or difference) between successive LBA requests using (1). In the rest of this article, the difference between successive LBA requests is abbreviated as "LD"

$$LD_t = \mathrm{LBA}_{t+1} - \mathrm{LBA}_t. \tag{1}$$

Both LBA_{t+1} and LBA_t indicate the accessed LBA at time $t+1$ and t, respectively, and LD_t indicates the LD pattern between t and $t + 1$. For example, as shown in Table I, two different sectors of LBAs (as shown in columns 2 and 5, respectively) are selected from the trace and the LD pattern of each sector is calculated (as shown in columns 3 and 6, respectively). In this example, the two selected sectors have the same LD pattern and thus the latter sectors of LBAs can be predicted based on the prior sectors of LBAs. However, while encountering the LBA requests shown in column 5, existing data prefetchers are hard to predict and prefetch the correct data with only considering the known LBA patterns in column 2. This new LD feature can not only help on improving the learning accuracy of the learning-based prefetcher but also reduces the learning complexity.

IV. Our Approach: DeepPrefetcher

A. Prefetching as Supervised Learning Problem

To locate a specific block in the device, the SSD controller uses a virtual mapping strategy to translate the LBA to physical block address (PBA). This virtual mapping has numerous benefits. Wear leveling tasks, for example, would shift the location of data to reduce even wear leveling on the memory cells and thus enhance the endurance of flash cells [18]–[22]. Hence, to keep track of each block, the mapping table will be updated to maintain the read latency. Therefore, treating the prefetching task as the regression problem will lead to a wrong prefetch because the regression algorithm produces a continuous-valued output, whereas the LBA value is finite.

Relying on this ground, we opt to treat prefetching as a supervised learning problem. In supervised learning algorithms, a model accepts a set of inputs, denoted by X with the corresponding output, which is denoted by y and learns to get the best possible mapping function f with a dataset of input–output pair and use it to make predictions in the future inputs (i.e., $\hat{X}$) that have unknown outputs (i.e., $\hat{y}$). As supervised learning algorithms require the input and output pair to learn the mapping function, the *LBA Preprocessor* algorithm is proposed in this work. As shown in Algorithm 1, the *LBA Preprocessor* accepts a long sequence of past observed LBA access patterns (i.e., T) and change it to a subsequence of $\{X, y\}$ pair. The overall idea of this algorithm is to use the first l LBA sequences (i.e., $T_0, T_1, T_2, \ldots, T_{L-1}$) as the input and use the next value T_l as the output (or class label y) of prediction. The window size L represents the total number of LBAs covered by each time step to form the input–output pair. By applying the *getSubsequence* function in each iteration, the LBA sequence covered by the rolling window L will be placed

TABLE II
CONVERTING LBA ACCESS PATTERNS TO SUPERVISED LEARNING PROBLEM

X1	X2	X3	X4	...	y
31814798	31814982	31822014	1478311	...	19667566
31814982	31822014	1478311	19667566	...	13143533
31822014	1478311	19667566	13143533	...	34024427
1478311	19667566	13143533	34024427	...	33979196

in the input (step 7) and output pairs (step 8). Furthermore, the value of L will constrain the minimum information required to perform the inference. This is because, as the value of L increases, the model size increases; subsequently, the prefetching latency will increase. At the end of each iteration, the index of T will move forward by k sampling intervals (step 9) based on the value of overlap percentage O, and this value can range from 0% to 100%. Note that when a smaller overlap percentage O is applied, the window moves a larger distance forwards after each cycle and thus generates fewer LBA subsequences; as a result, it incurs the phenomenon of *underfitting*, which makes the prefetching model (see Section IV-B) fail to learn the important features for inference. On the other hand, if the value of O is high, the algorithm generates a high number of subsequences. However, since the similarity between two consecutive subsequences is very high, it incurs the phenomenon of *overfitting*, which makes the prefetching model fit too close to the training data and tends to fail in predicting the testing/inference data. Thus, to have a reasonable tradeoff, the DeepPrefetcher prototype is implemented using a 50% overlap and the structure of the LBA sequence after preprocessing using Algorithm 1 is shown in Table II. In Table II, the columns labeled as x represent the input sequences, whereas the one labeled as y denotes the output or class label (i.e., the next LBA to be prefetched).

After preprocessing time series of LBA access sequence using Algorithm 1, the next LBA to be prefetched (i.e., LBA_{t+1}) can be formulated as

$$\text{LBA}_{t+1} = \hat{f}\big(\text{LBA}_{t-l}, \text{LBA}_{t-(l-1)}, \ldots, \text{LBA}_t, \theta\big) \tag{2}$$

where $\hat{f}$ is the desired function supposed to be a block access pattern prediction model, l is the rolling window length (or the number of observation to look back), and θ is the set of parameters for the function $\hat{f}$. To take the advantage of differencing, (2) can be rewritten as

$$\text{LBA}_{t+1} = \text{LBA}_t + \hat{f}\big(LD_{t-l}, LD_{t-(l-1)}, \ldots, LD_{t-1}, \theta\big). \tag{3}$$

In another word, the value of LBA at time $t+1$ is equivalent to the addition of LBA value at time t and the predicted LBA difference (LD_t) using function $\hat{f}$ [see (1)].

In recent years, deep learning models, such as recurrent neural network (RNN) and its variant, are proven to be a potential solution to capture the patterns in complex sequence structures, example, music generations [23] and machine translations [24] tasks. Unlike other neural networks, RNNs have a memory in their architecture that helps the model to remember long-term dependence on time-dependent sequential data. Furthermore, it has been proven that RNN is a universal approximator provided that the network has enough data for training and has an activation function in the hidden units [25]. This concept makes RNN and its variants applicable to data prefetching tasks.

B. Method

In this section, we present a data prefetching strategy called "DeepPrefetcher," *which is a deep learning approach to predict and prefetch data from* NAND *FLASH to SSD DRAM Buffer*. As shown in Fig. 3, DeepPrefetcher includes three main components, i.e., embedded representation, feature-learning, and feature-classification. First, a series of LBA differences ($LD_1, LD_2, \ldots, LD_{t-1}$) in the training set pass-through an embedding layer containing a lookup table mapping them to dense vectors ($V_1, \ldots, V_k, V_k \in$R). This new representation is included in DeepPrefetcher to capture the contextual similarity and semantic relationship between LD sequences, which is essential for prefetching. These vectors are then handled by the LSTM to generate the representative feature vector representing the whole sequence. Following that, the fully connected layer (i.e., FC) will perform an affine transformation on the learned feature to perform classification with a *Softmax* function [see (9)].

1) Embedded Representation: At its most fundamental level, a machine learning model is all about representation learning. In this way, utilizing the LD sequence [i.e., the prepossessed LD value using (1)] as a discrete id and passing into the deep models lead to data sparsity, which prompts a challenge to capture the hidden representation of the series. Thus, to avoid the dominance of LD with big value during prefetching/classification, they are encoded in a way that improves the prediction performance. There are two possible ways to encode the LD sequence: 1) one-hot representation and 2) embedded representation.

In *one-hot encoding*, each potential values of a sequence are encoded to a vector of length T, where T is the total number of unique value that the input can take (which is equivalent to the number of valid LD in a DeepPrefetcher). For this work, the *embedded representation* (also called *word2vec* model [26]) is adopted due to the fact that it can solve the aforementioned problem of one-hot encoding representation by mapping each input in a sequence into a new distributed representation that contains the neighboring (i.e., spatial locality information) of the input with the neural network. Rather than winding up each input with a T-dimensional vector, this strategy represents the input with a much smaller embedding matrix of size k, where the value of k decides how long we want the vector to be (or shrinking factor). It is important to note that the higher the value of k, the more semantic relation it encodes, whereas the smaller the value, the faster to train the model. To balance this tradeoff, most of the time, the embedding vector size is hundreds of dimensions. In DeepPrefetcher, the value of k is set to 512, which is likely several orders of magnitude smaller than the number of possible LD (or LBA) values in our dataset. The context window controls the number of LD to look back and forward in a sequence (or called window size). Suppose that the input LD sequence is $[2, 2, 4, 8, 6]$, and the window size is 1 (i.e., neighbors). For $LD = 8$, the context matrix (i.e.,

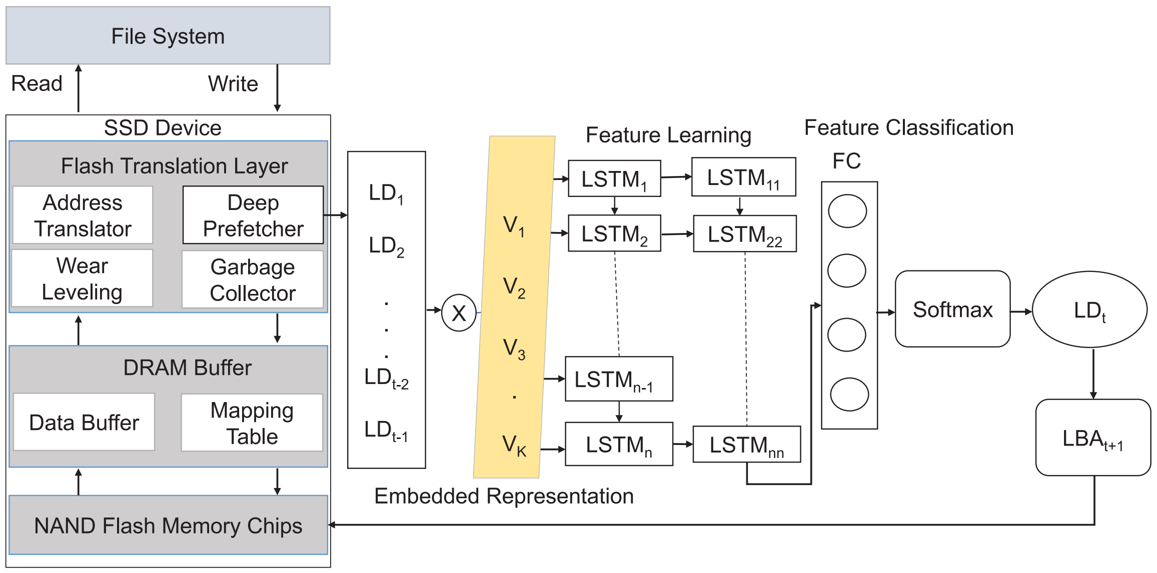

Fig. 3. Overall architecture of DeepPrefetcher framework.

a set of LD within a window size of 1) becomes [4, 6]. The context matrix is constructed by iterating the entire LD values of each sequence in a training dataset. During the training, the model learns to capture the hidden connection between a given LD and its corresponding context matrix and represent it in the embedding vector of the structure $[v_1, v_2, \dots, vk]$, where k is the dimension of the embedding vector. Note that those weights (i.e., v) start as random values, and the model is then trained in order to adjust the weights to represent each LD to a corresponding embedded representation. The value of v (or weights) in an embedded vector indicates how similar a given LD_i is to the other LD_j values in the training dataset. In other words, if two LD values frequently appear together in the training data, the model produces likely similar embedded representation for both LDs. Thus, the input to our DeepPrefetcher LSTM is the embedded vector that is rich in context compared to the real value of LD. As reported in [27] and [28], these transformations are usually pretrained and taken as the input for the deep model. In contrast to this strategy, we train the embedding's with the other part of the network, which empowers the model to update the embedding weights during the training phase by back-propagating the total error of loss function to the embedding layer. The utilization of this strategy is invaluable because it enables the model to tune the embedding weight during the training.

In general, this distributed representation has two implications to DeepPrefetcher: 1) it reduces the dimension of the input into a k-size vector and 2) it is a more expressive representation as it captures contextual resemblance and semantic sequence of data.

2) Feature Learning: In our design, we are going to have our network learn how to predict the next LD (i.e., LD_t) given the prior LD sequences [i.e., $(LD_1, LD_2, \dots, LD_{t-1})$]. Achieving this prediction task requires a recurrent neural design since the network will have to remember the input at each time step to capture the context, which is important for the prediction. For instance, to predict the output of the

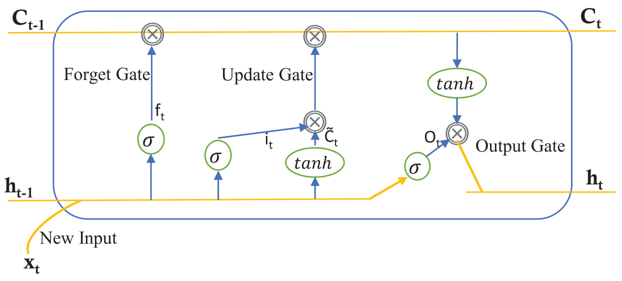

Fig. 4. LSTM cell structure (the variable t indicates the times steps which are the past observation).

LD sequence [2, 2, 4, 6, ?], the model has to know the context of the sequence to infer that the output is y; otherwise, relying solely on the last input will lead to incorrect inference/prefetching. This is because predicting a series requires the memorization of past inputs to infer the correct outputs.

Considering the access pattern prediction as a classification problem, the feature learning process in DeepPrefetcher is done by a stacked sequence of the LSTM layer. The choice of LSTM is due to the fact that LSTM has an inner cell state that allows information to flow across layers (that can represent the context information) so that suitable data can be transmitted down the lengthy sequence chain to generate predictions [29]. The detail structure of each LSTM cell utilized in DeepPrefetcher is shown in Fig. 4, which consists of a hidden state h_t and a cell state C_t, together with input gate i_t and forget state f_t. Once the current input x_t, and the previous hidden state h_{t-1} is provided to the LSTM, the final output is calculated as follows.

1) Decide what data to discard from the incoming cell state using a sigmoid function. The output 1 indicates this data is kept entirely, and 0 to mean discard everything. This is conveyed in mathematical terms as follows:

$$f_t = \text{sigmoid}\big(W_f \cdot \big[h_{t-1}, x_t\big] + b_f\big) \quad (4)$$

where f_t is forget-gate value, and b is a bias term.

2) Determine what information to update. This has two parts: first, decide the values to update(i_t) using sigmoid function, and then create a candidate vector $\tilde{C}_t$ of new input using tanh function and combine it with the sigmoid function to produce a new cell state C_t (or memory). These update process are provided mathematically by

$$\begin{aligned} i_t &= \sigma\left(W_i \cdot \left[h_{t-1}, x_t\right] + b_i\right) \\ \tilde{C}_t &= \tanh\left(W_c \cdot \left[h_{t-1}, x_t\right] + b_c\right) \\ C_t &= f_t * C_{t-1} + i_t * \tilde{C}. \end{aligned} \tag{5}$$

3) As appeared in Fig. 4, the LSTM cell has two outputs, i.e., hidden-state (h_t) and cell-state (C_t). In LSTM, the cell state is an internal memory of the LSTM that acts as an information pathway to the next LSTM, whereas the hidden state carries a learned vector representation about what has been seen so far, and it is calculated as follows:

$$\begin{aligned} o_t &= \sigma\left(W_0 \cdot \left[h_{t-1}, x_t\right] + b_0\right) \\ h_t &= o_t * \tanh(C_t) \end{aligned} \tag{6}$$

where o_t is the output-gate. Furthermore, to introduce nonlinearity to the network, the output of hidden states will pass through a rectified linear unit (ReLU) function which is given as follows:

$$ReLU(h_t) = \begin{cases} 0, & \text{if } h_t < 0. \\ h_t, & \text{otherwise} \end{cases} \tag{7}$$

where the output of this function is 0 for all negative values, and h_t for all positive values. By rehashing the entire process until the final LSTM layer is reached, the model learns to capture the required feature to perform the prediction.

Overall, the final hidden state output (i.e., feature learning process output) carries the learned vector representation of information about what LD sequence an LSTM cell has seen over the time and pass it to the classification layer for inference.

3) Feature Classification: Following the feature-learning process, the next LBA prediction (i.e., feature classification) process involves three steps. First, the output of feature learning process (i.e., LSTM) will undergo an affine transformation in the fully connected layer (i.e., FC) to learn the aggregate information required to map the input to produce the number of outputs, and it is given as follows:

$$\theta = W^T h_t \tag{8}$$

where W is the weight of the hidden layer, and h_t is the output of LSTM. It is important to note that the number of hidden neurons in the fully connected layer (i.e., FC) is equivalent to the total number of unique LD in the training data. Next, the *softmax* function is applied to the output of (8) to produce a likelihood score for each possible LD that the model is attempting to predict, and is calculated as:

$$P\left(y = j|\theta^i\right) = \frac{e^{\theta^i}}{\sum_{j=0}^{k} e^{\theta^i}} \tag{9}$$

which can be read as the normalized probability of output ($y = j$) given the previously hidden state value (θ). Finally, the candidate LD that maximizes the likelihood is chosen as the output. But, as the LD value is not the final task of the proposed prefetcher, the final output LBA to be prefetched is calculated using (3). Note that the number of LBA to prefetch at once in DeepPrefetcher is flexible (i.e., without changing the model structure, we can permit the model to predict a variable number of LBA sequences). Let say the number of LBA to prefetch at a time is set to n, and the provided LD sequence is [2, 2, 4, 6]. First, the model yields a single output y_1, and then ignore the oldest input and concatenate the remaining inputs (i.e., 2, 4, 6) with y_1 to produce y_2, and this process continues until the final value is prefetched. Besides, using the output of (9) as a confidence score, the model balances the prefetcher coverage and accuracy. For instance, during aggressive prefetching (i.e., $L > 2$), if the probability score associated with the first predicted LD is higher than the second predicted LD by a predefined threshold, the prefetcher is altered to be less aggressive to overcomes the impact of polluting the buffer with the wrong data. The aggressiveness is increased to the maximum setting (maximum value of each experiment) when the confidence score keeps increasing or (higher than the expected difference or threshold). Therefore, by dynamically controlling the prefetch length (or aggressiveness) based on the confidence score produced by the model, the proposed model can not only achieve higher coverage but also eliminate unnecessary prefetch to achieve high accuracy.

4) Model Training and Optimization: All the details discussed up to this point involves how a single LD sequence passes through the network to generate the final LBA as the output. However, neural network training includes multiple-passing through the network to calculate the loss function and update the parameters. Among the available alternatives, DeepPrefetcher utilizes categorical cross-entropy loss, which is the often used loss function for the multiclass classification problem [30]. Given the actual value of a sequence (i.e., y), and the predicted value (i.e., $\hat{y}$), the prediction errors produced by the model after each batch of training is calculated using categorical cross-entropy function as follows:

$$\text{loss}\left(y, \hat{y}\right) = -\sum_{i=0}^{M}\sum_{j=0}^{N}\left(y_{i,j}\log\left(\hat{y}_{i,j}\right)\right) + L2 \tag{10}$$

where $\hat{y}_{i,j}$ is the normalized probability output by the model, M denotes the number of unique LDs in the training data, N represents the number of data in a batch of training, $L2$ is the regularization term used to penalize the large weight to minimize the complexity of the model, and $y_{i,j}$ is the binary value which indicates whether i is a correct class or not. The output of this function will help us to assess how well the trained model works on a provided data, i.e., if the output deviates from the real outcomes, the value of loss function would be high. The optimal loss (or the desired accuracy level) can be reached by repetitive training using the optimization algorithms, which spread the error back from the last layer to all the prior contributing layer backward. This process repeats until the minimum loss function is reached. To be specific, we use Adam optimizer [31] for this work, which enables DeepPrefetcher to learn the learning rate for each parameter adaptively and efficiently. Furthermore, as a means to reduce the impact of overfitting, some neurons are arbitrarily dropout during the training phase. In addition to handling the overfitting problem, the addition of a dropout strategy will also produce a generalized model, that is, an ensemble effect of numerous neural networks [32]. Finally, once the optimal loss is found, the model is saved as a pickle file, which is later used for inference (i.e., for the experiment part).

Besides, the lookup time is also one of the design concerns in cache-based system design. In this work, the DRAM buffer was implemented using least recently used (LRU) cache with a hash map of keys and a double linked list. The usage of a hash map allows the fast lookup with $O(1)$, whereas the usage of double linked nodes make the insertion and removal operations to be $O(1)$ [33].

Overall, the newly proposed prefetcher is generally distinct from the frequently used sequential prefetcher and other schemes. It abstracts the semantic relationship of block access using the embedding vector and uses this vector to model the long-term dependencies among different blocks to make an inference. As a result, it not only captures the irregularity of access patterns and increases the buffer hit ratio but also increases the model's generalization ability to perform the inference on unseen LBAs.

V. Performance Evaluation

A. Evaluation Metrics

Intuitively, the prefetching algorithm can either leads to a buffer hit or miss. A buffer hit occurs when the program uses the prefetched data before they are removed from the buffer, and there is a buffer miss when the program does not use the prefetched data ahead of their removal from the buffer. In such a way, the primary metrics used to evaluate the prefetching algorithms are accuracy, coverage, and timeliness [34]. They measure, respectively, the ratio of relevant prefetches to total issued prefetches, the number of misses conquered by the prefetching algorithm, and how soon a block is prefetched versus when it is referenced. It is necessary to optimize these metrics together. For instance, by naively elevating the amount of prefetcher, high coverage can be attained, but this could pollute the buffer by increasing the unwanted traffic and lead to decay in performance. Therefore, high accuracy value, low latency, and high coverage are required.

Accordingly, the two evaluation metrics (accuracy and coverage) are calculated as follows:

$$\text{Accuracy} = \frac{\texttt{prefetch\_hits}}{\texttt{total\_prefetch}}$$
$$\text{Coverage} = \frac{\texttt{prefetch\_hits}}{\texttt{misses\_without\_prefetching}}. \quad (11)$$

Evaluating the timeliness of each prefetch is also essential to fully assess the efficiency of a prefetching algorithm. That is, if the prefetch began far enough in advance of the miss, all the punishment associated with the miss could be avoided. However, if the prefetch only began one or two steps ahead of the request, there may still be a significant amount of lag associated alongside the prefetch. To assess this delay correctly, we need to have a more comprehensive processor and memory system model, which is beyond the scope of this work.

In this work, we assess the performance of prefetching in terms of the speedup gain by the prefetching strategies, which is calculated as follows:

$$\text{Speedup} = \frac{(\text{Service time}_B - \text{Service time}_P)}{\text{Service time}_B} \times 100 \quad (12)$$

where B and P denote the service time for a system with no prefetcher and a system with prefetcher, respectively. The service time is the time required to serve read requests, and it is calculated as follows:

$$\text{Service time} = h \times x + m \times y \quad (13)$$

where h and m are the numbers of DRAM hit and DRAM miss, respectively, whereas x and y denote the cost of reading data from DRAM buffer (in microseconds) and the cost of reading data from NAND flash, respectively. As shown in Fig. 1, whenever the program needs to read the SSD data, the SSD controller first looks to see if the requested data is in the DRAM buffer. In this case, the data is directly served from the DRAM buffer, avoiding the need to go down further. If not, a buffer miss occurs, and the request must go to the NAND Flash to fetch the data. The latter alternative can be on the order of 10 to 100 times slower than the first one. Thus, the service time increases as the number of DRAM miss increases.

Apart from deciding which LBA to prefetch, a prefetch algorithm must choose where to hold the fetched data and the frequency with which a prefetch can be pursued. In our simulation, the prefetched data will be placed in the DRAM buffer of SSD. For all studies, we consider LRU policies to place and displace data to and from the DRAM buffer. During prefetching, when there are no free buffers, the buffer with the oldest prefetched LBA is selected for replacement, and the current prefetched data will be written to the beginning of the buffer.

In addition to the placement strategy, the recurrence with which a prefetch is sought is an important design parameter since each prefetching endeavor a lookup to check whether the data is already in the buffer or not. First, frequent prefetching can flood the buffer and defer access to the requested one. By contrast, a rare prefetching on buffer miss can decrease the coverage of the prefetcher. Prefetching on each block reference, for example, can twofold the traffic, while prefetching only on buffer misses can upper limit the buffer misses coverage. To have a reasonable comparison with the other schemes, the experiment in this article uses prefetch on miss strategy. The prototype used for the experiment is developed using Python and Keras deep learning framework.

B. Evaluated Prefetchers

In addition to the proposed model, we simulate four more schemes (see Section II-B for details) that target disk access pattern for comparisons and they are summarized as follows.

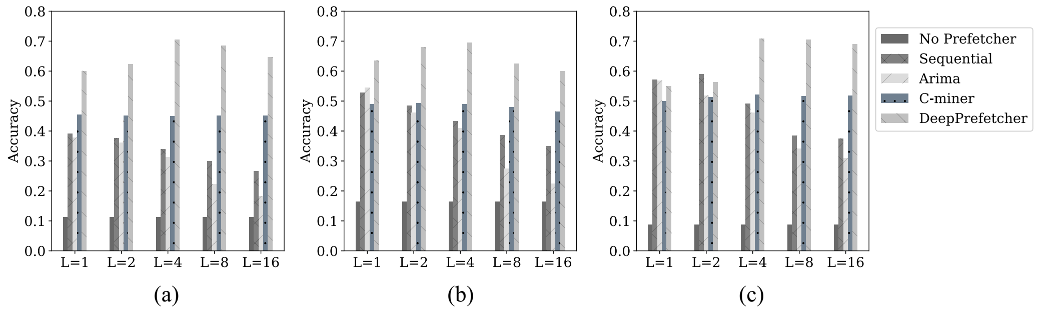

Fig. 5. Prefetching accuracy analysis using different models and different prefetch length (L). (a) Media Server Trace File (MSTF). (b) Print Server Trace File (PSTF). (c) User Home Directory Trace File (UHTF).

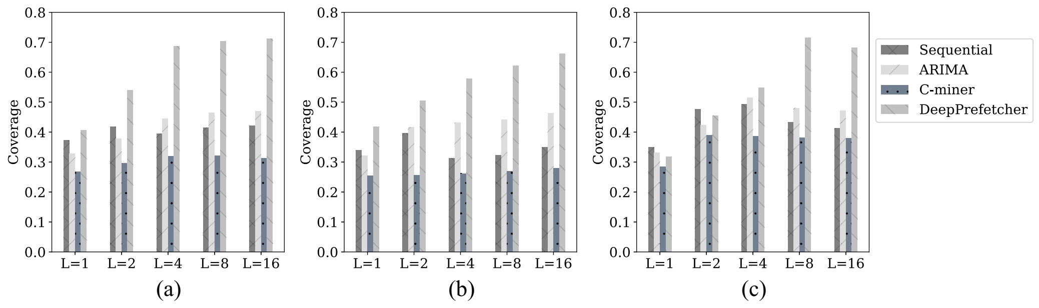

Fig. 6. Prefetching coverage analysis using different models and different prefetch length (L). (a) Media Server Trace File (MSTF). (b) Print Server Trace File (PSTF). (c) User Home Directory Trace File (UHTF).

1) *No Prefetcher:* This simulates SSD without any I/O prefetching facilities and it was used as a baseline to demonstrate the gain by other schemes.
2) *Sequential Prefetcher:* The rule that guides prefetching in this scheme is constant, which is prefetch LBA_{t+1} whenever LBA_t is requested.
3) *ARIMA Model:* This scheme uses model built at time t to forecast data that can be requested at time $t+1$.
4) *C-Miner:* It mines the frequent association rules from the training dataset to derive block correlations for prefetching.

C. Dataset and Evaluation Setups

We used the methodology of assessment outlined in [35], which proposes making trace-driven simulations with several disk traces gathered in actual systems. For this purpose, we create an SSD simulator with an LRU replacement policy to emulate a real storage system with 500 GB of storage and 16 MB of DRAM buffer. The DRAM buffer uses a 4KB block size to match the NAND flash 4KB page.

To show the potential and adequacy of the proposed data prefetching schemes, trace-driven I/O simulation experiment was conducted using MSRC block I/O trace files shown in Table III. Since the MSRC block, I/O trace covers different domains it was regarded as the representative trace file for block pattern analysis [17].

In this work, the standard leave-one-out cross-validation (LOOCV) technique is used for the model evaluation. The main idea of LOOCV is to utilize $n-1$ traces or (samples) for training the model and conduct the experiment on the remaining one. We opt to use this approach because it constitutes an unbiased estimate of the true generalization indicator or model performance [36]. Note that the C-miner and DeepPrefetcher schemes are trained offline on a training set, and the model is saved as a pickle file for later use during the prediction (prefetching). As the ARIMA scheme needs remodeling for each new observation, we train it online for each observation. For the remaining schemes, retraining the model is not required.

TABLE III
DATASET SUMMARY

Server	Function	Trace File	Total Read
mds	Media Server	mds_1	996318
usr	User Home Directories	hm_1	591627
prn	Print Server	prn_0	344057

As shown in Figs. 5 and 6, we force each model to predict and prefetch the variable number of blocks (i.e., 1, 2, 4, or 16) for each trace file.

D. Prefetching Analysis

This section reports the prediction coverage and accuracy results of the DeepPrefetcher and baseline models on different benchmarks.

The substantial performance of the proposed model compared to the other prefetcher is shown in Figs. 5 and 6, where the *x*-axis in both figures indicates the prefetch degree (L), and the *y*-axis shows the performance results in terms of accuracy and coverage. As presented in Fig. 5, the proposed scheme had a higher average accuracy than the remaining methods in all trace files. The average accuracy of sequential prefetcher, ARIMA model, C-miner, and DeepPrefetcher are 41%, 37%, 48%, and 64%, respectively. Among the others, prefetching with the ARIMA model was the least in terms of accuracy (i.e., 37%) in all trace files. Two factors contributed to this outcome. In the first place, the ARIMA model treat the LBA access pattern as a regression problem, and accordingly, it produces a continuous-valued number which prompts to incorrect LBA access. Second, the ARIMA model requires long stationery series to make a right prediction, however, as referenced in Section III, the I/O access patterns are sporadic and, hence, it leads to the wrong prediction. As depicted in Fig. 6, similar results have been noted in terms of coverage metric, where the coverage is computed by diving the count of helpful prefetches by the count of cache misses without prefetching. The evaluation result demonstrates that the proposed model improves the coverage rate of Sequential, ARIMA, and C-miner by 17%, 14%, and 25%, respectively. The coverage rate of C-miner is low (31%) compared to the other schemes. In the C-miner, the rule governing the prefetcher is the association rules derived from the training data. Thus, when used to predict infrequent access patterns, the C-miner will not issue prefetching at all; thus, the C-miner coverage rate is low compared to the other schemes.

On the other hand, confidence and aggressiveness of the prefetcher determine the performance benefit and the damage brought by the model. For instance, by naively increasing the prefetch length (or aggressiveness) on a buffer misses, the coverage will increase; however, it does not always improve prediction accuracy because the model brings more data that are L distance away from the recent accessed address. Thus, to validate the tradeoff between aggressiveness and prediction confidence of the proposed model and other baseline models, we experiment with a less aggressive setting ($L = 1$) to a more aggressive setting ($L = 16$). Those settings are the maximum limit to prefetch at a time for each experiment on a buffer miss. Sequential and ARIMA models have the most severe tradeoff between the prefetch degree and the accuracy metrics. With a prefetch degree of 1 or 2, they have higher accuracy scores, but the accuracy keeps decreasing as the degree of prefetching increases to 16. For instance, in a UHTF trace file [Fig. 5(c)], the sequential prefetcher will exploit when the length of prefetching is 1 or 2, and this result starts to diminish as the length of block increases to 4, 8, and 16. For sequential prefetcher, whenever there is a miss to consecutive access, the model prefetches L continuous blocks without having the opportunity to check for the intermediate result. That means if the value of $L = 16$, it always prefetches 16 continuous blocks whenever there is a miss. Thus, since there is no feedback loop for an intermediate checkup, it pollutes the buffer with incorrect data; as a result, the accuracy drops as L increases. Likewise, the ARIMA model tends to work well for short-term prefetching (i.e., when the value of L is smaller), but not for longer-term forecasts (when the value of L increases). One major issue is that the ARIMA model depends on the previously computed regression coefficient and accumulated errors to keep forecasting for the next L consecutive LBA_s, and thus not effective at long-term prefetching. For instance, suppose the model is fitted as $\text{LBA}_t = \alpha + \beta_1(\text{LBA}_{t-1}) + \beta_2(\text{LBA}_{t-2}) + \epsilon$ (where β is the coefficient of regression, α is the intercept term, and ϵ is the autoregression error), and the value of $L = 8$. The model forecasts the value of LBA at time t (LBA_t) using the *LBAs* value at the previous two-time steps (or lags). To predict LBA_{t+1}, LBA_{t-2} will be replaced by LBA_{t-1} and LBA_{t-1} will be substituted by LBA_t, but that generally results in unrealistic forecasts as the length of L increases. In this way, the ARIMA model accuracy tends to be unstable (or decrease) with respect to changes in observations and changes in prefetch length. Unlike the other schemes, C-miner exhibit relatively stable result in terms of accuracy and coverage metric under different aggressiveness control setting. The main reason is that the C-miner does the prefetching only when the ongoing request matches any of the rules created during the training. Consequently, only a few prefetch rules generated during the training can be matched at runtime, while many generated rules ignored due to low confidence and support count. As a positive side effect, the accuracy and coverage rate will not change by changing the prefetch length.

On the contrary, the accuracy and coverage of DeepPrefetcher exhibit relatively an increasing trend across all the benchmarks under different aggressiveness setting, despite a slight drop after $L = 8$. There are different factors attributes to this performance. First, the stunning performance observed through DeepPrefetcher schemes is due to the iterative optimization algorithm used to find the optimal model parameter that learns to generalize on the unseen dataset with minimal error. The usage of the LSTM structure permits the model to remember the past data to make an inference on the new input sequence with the help of a feedback mechanism (see Fig. 4). Besides, the utilization of embedded representation for context learning allows the model to learn high-level and lower features from data in an incremental manner that is difficult to capture using sequential prefetcher. Furthermore, DeepPrefetcher utilizes a recursive look-ahead strategy to prefetch L subsequent blocks on a buffer miss. For a prefetch length L (aggressiveness setting), DeepPrefetcher predicts the first candidate LD_i and recursively appends the newly predicted LD_i with the recent LD sequence to predict the next LD_{i+1}. As discussed in Section IV-B3, the last layer (i.e., Softmax) in DeepPrefetcher outputs the probability score associated with all the possible outputs. Using this value as a confidence score, the model balances the prefetcher coverage and accuracy. During aggressive prefetching (i.e., $L > 2$), if the probability score associated with the first predicted LD is higher than the second predicted LD by a predefined threshold, the prefetcher is altered to be less aggressive to overcome the impact of polluting the buffer with wrong data. The aggressiveness is increased to the maximum setting (maximum value of each experiment) when the confidence score keeps increasing or (higher than the expected difference or threshold). In addition to checking the confidence score for prefetching, the model also checks if the subsequent prefetched LBAs are sequential. If the initial two anticipated LBAs are successive, the rest of the blocks will be prefetched sequentially; otherwise, they are prefetched using the DeepPrefetcher recursive look-ahead strategy. Therefore, as far as the confidence score of the predicted LD is high, prefetching more LBA will bring more

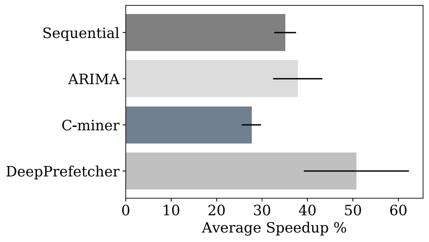

Fig. 7. Speedup of selected benchmarks with different prefetching schemes and prefetch lengths.

data to the buffer and thus increase the chance of buffer hit in the subsequent request. Besides, we also observe that a prefetch distance of eight provides superior results for DeepPrefetcher, albeit with significant variation between workloads. Overall, by dynamically controlling the prefetch length (or aggressiveness) based on the confidence score produced by the model, the proposed model can not only achieve higher coverage but also eliminate unnecessary prefetch to achieve high accuracy. From Figs. 5 and 6, we can see that the test results vary slightly across the trace files. The inconsistency in the accuracy and coverage emanates from the variation in storage size, layout, and application issuing the request across the traces. When working with machine learning models, it is undoubtedly the case that testing the trained model on another dataset yield a different result if the difference in the distribution of the training data and test data is high. Albeit, this does not seem to be the primary issue for DeepPrefetcher. The reason is that DeepPrefetcher is trained numerous times and was able to converge to nearly the same level of accuracy. However, we also believe that these differences can substantially impact the overall predictive performance of the model. Thus, the interested researcher can explore how to integrate online learning capability to DeepPrefetcher so that the impact of variation in the input will be minimal.

Likewise, we assess the service speedup of the four prefetching strategies against a system with no prefetcher. As shown in Fig. 7, the average speedup by the sequential method is 35%, whereas, for ARIMA prefetchers, it is nearly 7% on average. The peak speedup of sequential model appears at $L = 2$ with a value of 38%, and this result starts to diminish as the prefetch length increases (see Figs. 5 and 6). The reason for the decreased speedup at a higher value of L is because there is no dynamic tuning of the prefetch aggressiveness. For instance, if the value of $L = 8$ and the consecutive cache miss is detected by sequential prefetcher, the model prefetches eight consecutive blocks without an intermediate checkup of the subsequent `LDs`. Therefore, as the value of L increases, the number of sequential reading increases which bring the wrong data to the buffer and thus decrease the speedup and accuracy of sequential prefetchers. While the C-miner is relatively accurate (48%) for the prefetches it issues, the speedup rate is the least compared to the other schemes (27.6% on average). C-miner essentially matched fewer rules at runtime due to static confidence and support rules, and thus it prefetches less data, which lowers the speedup of C-miner. The reductions in miss rate using DeepPrefetcher translate to reductions in reading latency up to 61% (50.7% on average) contrary to a system with no prefetcher under different prefetch length. Dynamic tuning of prefetching based on the context and probability score generated by the model enables the DeepPrefetcher to throttle prefetching for low performing (or random patterns), thus avoiding unnecessary prefetch caused by useless prefetches. In particular, this feature allows DeepPrefetcher to works well compared to the other schemes when the prefetch length keeps increasing.

In general, by having the option to identify both the consecutive access and branches, the proposed model accomplishes higher coverage and accuracy than the other models. The result demonstrates that the proposed approach is more promising than similar techniques. Besides, as the application level context is not provided in the utilized evaluation benchmarks, the experimental result reflects the adaptability of the model across different applications. This is because the model is unaware of the application making a request, and it is fully trained on the LD access pattern derived from a sequence of LBA access patterns.

VI. Conclusion

Aiming to enhance the accuracy and coverage of data prefetching in DRAM-based SSD, this article presented a *DeepPrefetcher*, which is a deep learning way to systematically conquer the SSD DRAM miss ratio by predicting and prefetching the future access pattern. By transforming the LBA sequence into a new distributed representation that is rich in context, *DeepPrefetcher* employs the word2vec model and LSTM architecture to capture the hidden feature in the input sequence, which is important for prefetching. The experimental result demonstrated that DeepPrefetcher gains a prefetching accuracy of 21.5%, coverage by 19.5%, and speedup up to 17.2% compared to the other strategies. In future work, we will make an in-depth study on applying reinforcement-assisted learning for prefetching. Besides, we will also consider a way to capture the high-level application context to enhance the prefetching performance.

References

[1] T. Mitra, C.-K. Yang, and T.-C. Chiueh, "Application-specific file prefetching for multimedia programs," in *Proc. IEEE Int. Conf. Multimedia Expo. (ICME) Latest Adv. Fast Changing World Multimedia (Cat.No. 00TH8532)*, vol. 1. New York, NY, USA, 2000, pp. 459–462.

[2] C. Li, K. Shen, and A. E. Papathanasiou, "Competitive prefetching for concurrent sequential I/O," in *Proc. 2nd ACM SIGOPS/EuroSys Eur. Conf. Comput. Syst.*, 2007, pp. 189–202.

[3] C.-F. Wu, Y.-H. Chang, M.-C. Yang, and T.-W. Kuo, "Joint management of CPU and NVDIMM for breaking down the great memory wall," *IEEE Trans. Comput.*, vol. 69, no. 5, pp. 722–733, May 2020.

[4] M. Saxena and M. M. Swift, "FlashVM: Virtual memory management on flash," in *Proc. USENIX Annu. Techn. Conf.*, 2010, p. 14.

[5] D. P. Bovet and M. Cesati, *Understanding The Linux Kernel: From I/O Ports to Process Management*. Beijing, China: O'Reilly Media, Inc., 2005.

[6] N. Tran and D. A. Reed, "Automatic ARIMA time series modeling for adaptive I/O prefetching," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 4, pp. 362–377, Apr. 2004.

[7] Z. Li, Z. Chen, S. M. Srinivasan, and Y. Zhou, "C-miner: Mining block correlations in storage systems," in *Proc. 3rd USENIX Conf. File Storage Technol. (FAST)*, vol. 4, 2004, pp. 173–186.

[8] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang, “DiskSeen: Exploiting disk layout and access history to enhance I/O prefetch,” in *Proc. USENIX Annu. Techn. Conf.*, vol. 7, 2007, pp. 261–274.

[9] T. Kimbrel and A. R. Karlin, “Near-optimal parallel prefetching and caching,” *SIAM J. Comput.*, vol. 29, no. 4, pp. 1051–1082, 2000.

[10] C. Gao *et al.*, “Constructing large, durable and fast SSD system via reprogramming 3D TLC flash memory,” in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2019, pp. 493–505.

[11] W. Anacker and C. P. Wang, “Performance evaluation of computing systems with memory hierarchies,” *IEEE Trans. Electron. Comput.*, vol. EC-16, no. 6, pp. 764–773, Dec. 1967.

[12] T.-F. Chen and J.-L. Baer, “Effective hardware-based data prefetching for high-performance processors,” *IEEE Trans. Comput.*, vol. 44, no. 5, pp. 609–623, May 1995.

[13] J. W. Fu, J. H. Patel, and B. L. Janssens, “Stride directed prefetching in scalar processors,” *ACM SIGMICRO Newsletter*, vol. 23, nos. 1–2, pp. 102–110, 1992.

[14] Y. Chen, S. Byna, and X.-H. Sun, “Data access history cache and associated data prefetching mechanisms,” in *Proc. ACM/IEEE Conf. Supercomput. (SC’07)*, Reno, NV, USA, 2007, pp. 1–12.

[15] J. Liao, F. Trahay, B. Gerofi, and Y. Ishikawa, “Prefetching on storage servers through mining access patterns on blocks,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 9, pp. 2698–2710, Sep. 2016.

[16] E. A. Shriver, C. Small, and K. A. Smith, “Why does file system prefetching work?” in *Proc. USENIX Annu. Techn. Conf. Gen. Track*, 1999, pp. 71–84.

[17] *MSR Cambridge Traces Kernel Description*. Accessed: Feb. 15, 2019. [Online]. Available: http://iotta.snia.org/tracetypes/3,

[18] Y.-H. Chang, J.-W. Hsieh, and T.-W. Kuo, “Endurance enhancement of flash-memory storage, systems: An efficient static wear leveling design,” in *Proc. 44th ACM/IEEE Design Autom. Conf.*, San Diego, CA, USA, 2007, pp. 212–217.

[19] Y.-H. Chang, J.-W. Hsieh, and T.-W. Kuo, “Improving flash wear-leveling by proactively moving static data,” *IEEE Trans. Comput.*, vol. 59, no. 1, pp. 53–65, Jan. 2010.

[20] M.-C. Yang, Y.-H. Chang, C.-W. Tsao, and P.-C. Huang, “New ERA: New efficient reliability-aware wear leveling for endurance enhancement of flash storage devices,” in *Proc. 50th ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Austin, TX, USA, 2013, pp. 1–6.

[21] C.-F. Wu, M.-C. Yang, Y.-H. Chang, and T.-W. Kuo, “Hot-spot suppression for resource-constrained image recognition devices with nonvolatile memory,” *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 11, pp. 2567–2577, Nov. 2018.

[22] Y. Di, L. Shi, S.-H. Chen, C. J. Xue, and E. H.-M. Sha, “$1 + 1 > 2$: Variation-aware lifetime enhancement for embedded 3D NAND flash systems,” in *Proc. 20th ACM SIGPLAN/SIGBED Int. Conf. Lang. Compilers Tools Embedded Syst.*, 2019, pp. 45–56.

[23] R. Jozefowicz, W. Zaremba, and I. Sutskever, “An empirical exploration of recurrent network architectures,” in *Proc. 32nd Int. Conf. Mach. Learn.*, 2015, pp. 2342–2350.

[24] I. Sutskever, O. Vinyals, and Q. V. Le, “Sequence to sequence learning with neural networks,” in *Advances in Neural Information Processing Systems*. Red Hook, NY, USA: Curran, 2014, pp. 3104–3112.

[25] B. Hammer, “On the approximation capability of recurrent neural networks,” *Neurocomputing*, vol. 31, nos. 1–4, pp. 107–123, 2000.

[26] X. Rong, “word2vec parameter learning explained,” 2014. [Online]. Available: arXiv:1411.2738.

[27] J. Mao, W. Xu, Y. Yang, J. Wang, Z. Huang, and A. Yuille, “Deep captioning with multimodal recurrent neural networks (m-RNN),” 2014. [Online]. Available: arXiv:1412.6632.

[28] M. Artetxe, G. Labaka, E. Agirre, and K. Cho, “Unsupervised neural machine translation,” 2017. [Online]. Available: arXiv:1710.11041.

[29] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.

[30] Z. Zhang and M. Sabuncu, “Generalized cross entropy loss for training deep neural networks with noisy labels,” in *Advances in Neural Information Processing Systems*. Red Hook, NY, USA: Curran, 2018, pp. 8778–8788.

[31] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” 2014. [Online]. Available: arXiv:1412.6980.

[32] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A simple way to prevent neural networks from overfitting,” *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1929–1958, 2014.

[33] K. Shah, A. Mitra, and D. Matani, “An O(1) algorithm for implementing the LFU cache eviction scheme,” *CiteSeerX*, vol. 1, pp. 1–8, 2010.

[34] J. Lee, H. Kim, and R. Vuduc, “When prefetching works, when it doesn’t, and why,” *ACM Trans. Archit. Code Optim.*, vol. 9, no. 1, p. 2, 2012.

[35] J. Liao and S. Chen, “Optimization of reading data via classified block access patterns in file systems,” *IEEE Access*, vol. 4, pp. 9421–9427, 2016.

[36] G. C. Cawley and N. L. Talbot, “On over-fitting in model selection and subsequent selection bias in performance evaluation,” *J. Mach. Learn. Res.*, vol. 11, pp. 2079–2107, Aug. 2010.

Gaddisa Olani Ganfure (Student Member, IEEE) received the bachelor’s degree in computer science and IT from Wollega University, Nekemte, Ethiopia, in 2010, and the master’s degree in computer science from Addis Ababa University, Addis Ababa, Ethiopia, in 2013. He is currently pursuing the Ph.D. degree with the SNHCC Program, National Tsing Hua University, Hsinchu, Taiwan, and Academia Sinica, Taipei, Taiwan.

His research interests include the storage system, AI-based cybersecurity, and NLP.

Chun-Feng Wu (Graduate Student Member, IEEE) received the B.S. degree from the Department of Computer Science and Information Engineering, National Central University, Taoyuan, Taiwan, in 2014, and the M.S. degree from the Department of Computer Science, National Tsing-Hua University, Hsinchu, Taiwan, in 2016. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan.

He serves in research and development alternative service with the Institute of Information Science, Academia Sinica, Taipei. His research interests include memory/storage systems, embedded systems, operating systems, and the next-generation memory/storage architecture designs.

Yuan-Hao Chang (Senior Member, IEEE) received the Ph.D. degree in computer science from the Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan.

He is currently a Research Fellow with the Institute of Information Science, Academia Sinica, Taipei, where he served as an Associate Research Fellow from March 2015 and June 2018 and an Assistant Research Fellow from August 2011 and March 2015. His research interests include memory/storage systems, operating systems, embedded systems, and real-time systems.

Dr. Chang is a Senior Member of ACM.

Wei-Kuan Shih (Member, IEEE) received the B.S. and M.S. degrees in computer science from National Taiwan University, Taipei, Taiwan, and the Ph.D. degree in computer science from the University of Illinois Urbana–Champaign, Urbana, IL, USA.

From 1986 to 1988, he was with the Institute of Information Science, Academia Sinica, Taipei. He is a Professor with the Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan. He has published over 130 articles in professional journals and conferences. His research interests focus on real-time system, wireless sensor networks, distributed file systems, embedded file systems, and energy issues pertaining to cloud computing.